SYNGRESS

华章程序员书库

程序员密码学

Cryptography for Developers

（加）Tom St Denis
（英）Simon Johnson 著 沈晓斌 译

业界密码学标准系列工具LibTom开发者力作
首本专门针对程序员的密码学导引

- 对称分组密码、单向散列函数、消息认证码算法、组合加密和认证模式、公钥密码等等。
- 保密性，完整性，认证和不可否认四种密码学目标的特例。
- 完整地讨论了大整数算术，公钥算法和高级加密。

机械工业出版社
China Machine Press

本书特色

• 这是一本针对那些需要学习密码学，同时还要安全、高效地实现密码学算法的程序员的书籍。

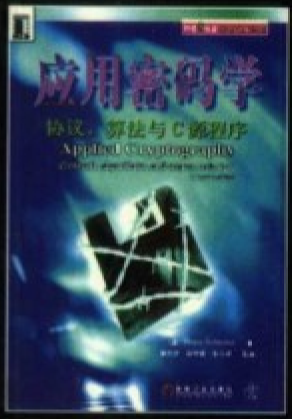
这本书从第1章介绍密码学研究领域开始。第2章全面论述了怎样实现RSA和ECC公钥算法所需要的大整数算术。后面的章节讨论了对称分组密码、单向散列函数、消息认证码算法、组合认证和加密模式、公钥密码和可移植编码实践的实现，保密性、完整性、认证和不可否认四种密码学目标的特例，以及公钥算法和高级加密。每一章都深层次地讨论了内存、大小以及速度性能上的权衡，也讨论了这些特定的主题都解决了哪些密码学问题。

本书还有一个与其相应的网站libtom.org，上面有关于实现多精度算术的超过300页的资料。

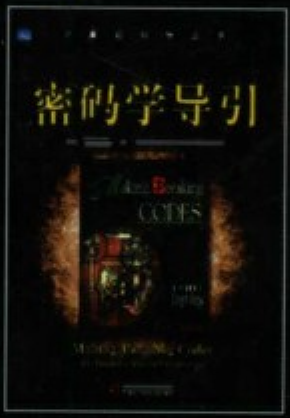
关于作者

Tom St Denis 是一名软件开发者，他因开发公共领域的LibTom系列加密算法而著名。在过去的5年时间里，他致力于宣传、开发和支持“开放源码的密码学”这一事业，并为其安全部署而努力工作。Tom目前受雇于Elliptic Semiconductor公司，为嵌入式系统设计和开发软件包。他和一个拥有各种硬件工程师的小组紧密合作，创建了一种BoB的硬件和软件组合系统。

Simon Johnson 是英国一名针对技术设备的安全工程师。Simon早在十几岁的时候就对密码学产生了兴趣，他研究了传统软件密码学的方方面面。他从17岁开始就活跃在密码学新闻组Sci. Crypt，参加了世界各地举行的各种安全会议，并积极促进安全计算实践。



ISBN 7-111-07588-9
定价：49.00元



ISBN 7-111-12478-2
定价：39.00元

上架指导：计算机/密码学

ISBN 978-7-111-21660-5

投稿热线：(010) 88379604
购书热线：(010) 68995259, 68995264
读者信箱：hzjsj@hzbook.com

华章网站 http://www.hzbook.com
网上购书：www.china-pub.com

封面设计·杨宇梅

ISBN 978-7-111-21660-5
定价：39.00元

版权声明

本书籍由pdf_world制作发布，针对目前图书数量繁多，质量参差不齐，读者面对买书易，买好书难的问题。pdf_world旨在为读者购书前提供另外一种快捷了解图书内容，挑选好书的方法。因此，pdf_world每周都会更新一批当年度最新的畅销书供读者选择。

核心订制书业务：只要是国内出版社出版超过半年的书，本店一般都可以为您制作扫描版电子文档，如有需要，请关注我们的官网：

http://pdf-world.taobao.com

声明：

1. 本书仅供读者购买纸本书前预览内容使用
2. 为尊重原作者的劳动成果，请于下载后48小时内删除
3. 如果您确实需要此书，请通过正规渠道购买纸本书

华章程序员书库

程序员密码学

Cryptography for Developers

(加) Tom St Denis
(英) Simon Johnson 著 沈晓斌 译

机械工业出版社
China Machine Press

这是一本针对软件开发人员而编写的密码学书籍，可帮助读者学习密码学，安全并高效地实现密码学算法。全书共9章，分别论述了ASN.1编码、随机数生成、高级加密标准、散列函数、消息认证码算法、加密和认证模式、大整数算术以及公钥算法等内容。书中每一章都深层次地讨论了内存、大小与速度性能上的权衡，也讨论了这些特定的主题都解决了哪些密码学问题。

Tom St Denis, Simon Johnson：Cryptography for Developers（ISBN: 978-1-59749-104-4）
Original English language edition published by Syngress Publishing, Inc.
Copyright © 2007 by Syngress Publishing, Inc.
All rights reserved.

本书中文简体字版由Syngress Publishing, Inc授权机械工业出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。
本书法律顾问 北京市展达律师事务所

本书版权登记号：图字：01-2007-1832

图书在版编目（CIP）数据

程序员密码学/（美）丹尼斯（Denis, T. S.），（美）约翰逊（Johnson, S.）著；沈晓斌译. —北京：机械工业出版社，2007.7
（华章程序员书库）
书名原文：Cryptography for Developers
ISBN 978-7-111-21660-5

Ⅰ. 程… Ⅱ. ① 丹… ② 约… ③ 沈… Ⅲ. 密码－理论 Ⅳ. TN918.2

中国版本图书馆CIP数据核字（2007）第089922号

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）
责任编辑：李南丰
北京牛山世兴印刷厂印刷 · 新华书店北京发行所发行
2007年7月第1版第1次印刷
186mm × 240mm · 21.25印张
定价：39.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换
本社购书热线电话（010）68326294

译　者　序

信息安全越来越受到人们的重视，而对信息安全的基石——密码学的研究也是如火如荼。但是，许多信息安全软硬件产品的开发者并不是专业的密码学研究人员。虽然他们擅长程序设计，而且现在也有不少各种密码学算法库，如LibTomCrypt和Crypto++，但在实现各种复杂的密码学算法时，由于缺少一定的密码学理论知识以及对密码学算法理解上的偏差，造成了对算法的使用不当，这又往往导致其开发的产品中存在各种潜在的漏洞及安全风险。同时，许多密码学理论工作者在实现密码学算法时，由于缺少程序设计技巧方面的知识，在算法实现的易用性和高效性上遇到了不少障碍，从而也会导致算法实现上存在不少安全缺陷。

本书的出版，在程序员和密码学研究人员之间架起了一座桥梁，使他们能够轻松地把理论和实践相结合。因此，本书也必将缓解信息安全业界的种种现实尴尬。本书的作者Tom St Denis既是一名密码学研究人员，也是一名开发人员，开发了各种软硬件安全产品，具有丰富的密码学相关产品的开发经验。同时，他自己也开发了Lib Tom系列密码学、数学算法库，如Lib TomCrypt、Lib TomMath、TomsFastMath。在这本书中，作者结合了Lib TomCrypt和LibTomMath等算法库，既介绍了实现各种密码学算法所必需的基本理论数学知识，也讨论了实现上的程序设计技巧；不仅有软件实现，也有硬件实现；不但从理论上分析了算法的安全性和性能，也从实现的角度讨论了如何正确并高效地使用各种算法，比如各种优化算法的实现，各种性能上的比较。

本书涉及密码学的各个研究方向，分组密码、散列函数、公钥密码以及相关的攻击，同时也讲解了密码学算法实现上所常用的ASN.1编码、大整数算术相关内容。这是目前市面上惟一一本把密码学算法的理论和实现结合在一起的书，也是惟一一本能够如此深入浅出地把这个方面融合到一起的一本书，没有深厚的程序设计功力和广泛的密码学理论知识是不可能写出这样一本书的。这无论对于需要开发安全产品的开发者，还是密码学相关研究人员来说，都非常值得一读，对安全产品开发工作以及密码学理论研究工作都起到相当的辅助作用，可以起到事半功倍的作用。

译者研究密码学数年有余，在国内知名的看雪软件安全论坛（http://bbs.pediy.com）任“软件调试论坛”版主（cnbragon），与论坛众多好手时常讨论密码学方面的问题，比较了解其在软件保护中的应用情况。译者也写了一个自己的加密算法库CryptoFBC。能够有机会把这样一本好的密码学应用书籍介绍给国内的广大程序员，译者感到非常荣幸。

在翻译本书的过程中，得到了许多人的帮助，尤其是北京华章图文信息有限公司的陈冀康编辑、看雪论坛的负责人段钢所给予的指导、建议与批评，使得译者受益匪浅，在此表示衷心的感谢！同时，我还要感谢iPB、RCT组织的成员以及看雪论坛上的其他朋友，他们总是在我最困难的时候给予无私和热情的帮助！我也要感谢我的导师，扬州大学信息工程学院殷新春教

授，他对我的关怀与鼓励，以及在密码学研究方向上的指导是我不断前进的动力！最后，我要感谢我的父母和我可爱的妹妹，他们总是给我无私地关怀、支持和理解！谨以此拙译献给所有帮助过我的人。

由于时间仓促，再加上译者水平有限，本书的翻译难免存在不妥甚至错误之处，敬请广大读者朋友不吝赐教和批评指正，译者深表谢意。

沈晓斌

2007年5月

http://bbs.pediy.com

前　　言

这是我的第二本书。读者朋友们，除了说这本书比上一本更加富有戏剧性并更加吸引人之外，我确实不知道该告诉你们什么。我不想解释过多的细节，但我只想说在MD5背后，RSA和SHA似乎有某些关系。下面让我们来认真地讨论正题吧。

当我写这篇前言的时候，本书将接近印刷，我期望看到最终的成品并且希望达到预期的目标。这本书是经过一年努力的结果，时间约为2006年初到2006年11月。我花了许多工作时间之外的业余时间来写作本书，目的就是能够更好地满足读者的要求。当然这也是一个很有趣的过程，虽然有时会很费力，但正如我的第一本书一样，是非常值得这么做的。

首先，在我深入讨论这本书之前，有必要介绍一下作者。这本书大部分是由我——Tom St Denis所写，并且在我的合著者技术审稿人Simon Johnson的帮助下共同完成的。我是一名来自加拿大安大略的计算机科学家，并且我对密码学相关的任何事情都很感兴趣。更加特别地是，我同时也是一名专用硬件和嵌入式系统开发的爱好者。

我想读者之所以知道我和我的这本书，大概是因为LibTom系列项目。这些项目是一系列的密码学以及数学相关的算法库，写这些库主要是用来解决现实中软件开发者会碰到的各种各样的问题。开发它们的同时也是为了能对读者有些指导作用。我的第一个项目，LibTomCrypt，是将近五年工作的结果，它支持很多有用的密码学基本函数，并且实际上也是这本书的一个很好的参考资源。为了继续完成密码学项目，我在2002年又开始了LibTomMath项目。LibTomMath是一个操作大整数的可移植数学库，已经成为LibTomCrypt项目中数学函数的默认提供者，同时也是其他项目（如Tcl和Dropbear）的一个组成部分。为了改进LibTomMath，我又写了TomsFastMath，它的速度非常快，并且也可以很容易地为密码学运算提供数学运算库。

我写的这些项目都是免费的，人们不仅可以直接免费获得，而且可以没有任何限制地使用。实际上它们都是完全公开的。至少对我来说，仅仅提供代码是不够的。我同时也提供了介绍如何使用这些算法库的文档。即使这样我认为还是不够。我还对代码进行了美化，同时也做了注释，这可以使这些代码本身具有一定的指导作用。第一个使用这种方式的是LibTomCrypt项目。2003年，我写了一本书，书名叫做*BigNum Math:Implementing Cryptographic Multiple Precision Arithmetic*的书（ISBN:1597491128），由Syngress出版社在2006年出版。这本书基本是把LibTomMath等项目里的代码直接插入到书中。这本书中同时也混有伪代码，它教会人们如何高效地操作大整数。

LibTom这些项目系列的完成得益于我在数年的开发过程中总结出来的座右铭：“开放的源码，开放的学术，开放的思想。”

其意思是说，通过提供有用的文档和参考资料的源代码，我们可以指导其他人并且使得他们产生新的想法和使用新的技术。由于这句座右铭具有指导意义，所以它扩充了原来的那种典

型的开放源码哲学。例如，GNU C编译器（GCC）的开放源码做法是很好的，但是它并不是一个教育项目。虽然这种想法已经谈得有点多了，但是我还将在我的下一本书（大概在2009年的某个时候出版）中以此为主题继续讨论它。

我继续开发我的LibTom项目，并且我也时刻提醒自己如果有可能的话，要尽力改进它们。我定期地参加如Toorcon这样的会议来传播“LibTom哲学”，以吸引新的开放源码开发者加入到这条教育战线上来。

那么Simon又是谁呢？Simon Johnson是一名来自英国的计算机程序员。他把他的时间都用在阅读和学习计算机安全和密码学技术上。更确切地说，他是一名开发C#应用程序的安全项目师。Simon和我相遇在sci.crypt用户新闻组，并且一起合作开发了各种项目。在整本书中，Simon充当了技术审稿人的角色。即使他很想给这本书的工程提供更多的帮助，但是他的时间安排不允许他这样做，然而他的帮助却又是那么的关键。可以确定地说，在以后的几年中，我们可以看到Simon会出版几本他自己的书。

本书讨论了什么？面向软件开发者的密码学。这种说法似乎是权威的并且不依赖于任何其他东西，但实际上是不全面的。对非密码专业人员开发者来说，这本书确实是一本必需的导引书籍。但是，这并不是说这是惟一一本关于这种主题的书籍。我们也经常把其他书作为参考书籍。毫无疑问，你需要一本“BigNum Math”，是介绍用于公钥算法中的大整数运算实现的必备书籍。另外一本必备书籍是*The Guide to Elliptic Curve Cryptography*（ISBN 038795273X），它很好地介绍了开发者所要了解的椭圆曲线密码学基本算法。出于对读者有益的角度考虑，我们主张读者看一些比较容易阅读的相关书籍。同时，你也可能需要阅读一些标准文档。例如，当你需要实现RSA加密体系时，确实需要一份PKCS#1拷贝（免费的）。虽然这本书也讲述了PKCS#1操作，但有一份标准文档也相当方便。最后，我强烈建议读者能够得到LibTom项目的拷贝，以获得第一手的密码学软件的使用经验。

本书是为谁准备的？我写这本书是为了那些针对我的项目，给我发电子邮件让我进行技术支持的人。但是这并不是说本书就是关于我的那些项目的，也不仅仅是讨论用户在使用这些库的时候会碰到的问题。通常，那些具有解决安全问题任务的软件开发者并不是专业的密码人员。他们都是聪明人，如果能够得到细心的指导，那么他们就可以实现安全的加密系统。本书的目的就是引导软件开发人员在其开发过程中如何去解决各种密码学相关的问题。如果你曾坐下来问自己：“我怎样来使用AES？”那么，这本书就是为你准备的。

本书不是为了那些需要可靠的密码学学术文章的人而写的。本并不是“应用密码学手册”，也不是“密码学基础”。简单地说，如果你并没有实现密码学算法的任务，也许这本书并不适合你。这是我在设计和写作这本书的时候所考虑的一部分。我们努力囊括足够的技术和学术理论上的细节，这可以让我们的讨论更加精确和有用。但是，我们省略了许多不适合这本书内容的密码学问题的讨论。

我想感谢在整个项目过程中帮助过我的每一个人。Greg Rose帮助我审校了其中一章。他也给我提供了一些灵感和很有见解的注释。我还想感谢Simon加入到这个项目并且为了本书的质量做出了很大的贡献。我同样也想感谢Microsoft Word给我吃的那些苦头。我感谢Syngress出版社的Andrew、Erin和其他为本书的最终付梓做出努力的人们。我也想感谢使用LibTom项目的

用户们，他们是这本书的灵感来源。没有他们的询问和分享他们的经验，我也绝不会首先写出这本书。

最后，我还要感谢那些预定了本书的读者朋友，由于出版日期的一再推迟而给你们造成了不便，对此我非常抱歉。

Tom St Denis

于加拿大安大略渥太华

2006年10月

PDG

目 录

译者序

前言

第1章 概述 …… 1

1.1 简介 …… 1

1.2 威胁模型 …… 2

1.3 什么是密码学 …… 3

1.3.1 密码学的目标 …… 3

1.4 资产管理 …… 7

1.4.1 保密性和认证 …… 8

1.4.2 数据的生命周期 …… 8

1.5 常识 …… 9

1.6 开发工具 …… 10

1.7 总结 …… 11

1.8 本书的组织结构 …… 11

1.9 常见问题 …… 13

第2章 ASN.1编码 …… 14

2.1 ASN.1概述 …… 14

2.2 ASN.1语法 …… 15

2.2.1 ASN.1显式值 …… 15

2.2.2 ASN.1容器 …… 16

2.2.3 ASN.1修改器 …… 17

2.3 ASN.1数据类型 …… 19

2.3.1 ASN.1头字节 …… 19

2.3.2 ASN.1长度编码 …… 21

2.3.3 ASN.1布尔类型 …… 22

2.3.4 ASN.1整数类型 …… 22

2.3.5 ASN.1位串类型 …… 23

2.3.6 ASN.1八位位组串类型 …… 24

2.3.7 ASN.1空类型 …… 24

2.3.8 ASN.1对象标识符类型 …… 24

2.3.9 ASN.1序列和集合类型 …… 25

2.3.10 ASN.1可打印字符串和IA5String类型 …… 28

2.3.11 ASN.1世界协调时类型 …… 28

2.4 实现 …… 29

2.4.1 ASN.1长度程序 …… 29

2.4.2 ASN.1原始编码器 …… 32

2.5 总结 …… 65

2.5.1 创建链表 …… 65

2.5.2 解码链表 …… 68

2.5.3 Flexi链表 …… 69

2.5.4 其他提供者 …… 70

2.6 常见问题 …… 70

第3章 随机数生成 …… 72

3.1 简介 …… 72

3.2 熵的度量 …… 74

3.2.1 位计数 …… 75

3.2.2 字计数 …… 75

3.2.3 间隙计数 …… 75

3.2.4 自相关测试 …… 75

3.3 它能有多糟 …… 77

3.4 RNG设计 …… 78

3.4.1 RNG事件 …… 78

3.4.2 RNG数据收集 …… 82

3.4.3 RNG处理和输出 …… 85

3.4.4 RNG估算 …… 89

3.4.5 RNG的设置 …… 91

3.5 PRNG算法 …… 92

3.5.1 PRNG的设计 …… 92

3.5.2 PRNG的攻击 …… 93

3.5.3 Yarrow PRNG …… 94

3.5.4 Fortuna PRNG …… 96

3.5.5 NIST的基于散列的DRBG …… 100

3.6 总结 ……104
3.6.1 RNG与PRNG ……104
3.6.2 PRNG的使用 ……105
3.6.3 示例平台 ……105
3.7 常见问题 ……107
第4章 高级加密标准 ……109
4.1 简介 ……109
4.1.1 分组密码 ……110
4.1.2 AES的设计 ……111
4.2 实现 ……121
4.2.1 一个8位的实现 ……122
4.2.2 优化的8位实现 ……127
4.2.3 优化的32位实现 ……129
4.3 实用的攻击 ……143
4.3.1 侧信道 ……144
4.3.2 处理器缓存 ……144
4.3.3 Bernstein 攻击 ……145
4.3.4 Osvik 攻击 ……146
4.3.5 挫败侧信道 ……146
4.4 链接模式 ……147
4.4.1 密码分组链接 ……148
4.4.2 计数器模式 ……151
4.4.3 选择一个链接模式 ……153
4.5 总结 ……153
4.5.1 荒诞的说法 ……156
4.5.2 提供者 ……157
4.6 常见问题 ……158
第5章 散列函数 ……161
5.1 简介 ……161
5.1.1 散列摘要长度 ……162
5.2 SHS的设计与实现 ……164
5.2.1 MD 强化 ……165
5.2.2 SHA-1的设计 ……165
5.2.3 SHA-256的设计 ……173
5.2.4 SHA-512的设计 ……180
5.2.5 SHA-224的设计 ……186
5.2.6 SHA-384的设计 ……187
5.2.7 零复制散列 ……188
5.3 PKCS #5 密钥衍生 ……189
5.4 总结 ……191
5.4.1 散列算法可以做哪些事 ……191
5.4.2 散列算法不能用来做哪些事 ……192
5.4.3 和口令一起工作 ……194
5.4.4 性能上的考虑 ……196
5.4.5 PKCS #5的例子 ……197
5.5 常见问题 ……199
第6章 消息认证码算法 ……202
6.1 简介 ……202
6.2 安全准则 ……203
6.2.1 MAC密钥的寿命 ……204
6.3 标准 ……204
6.4 分组消息认证码 ……204
6.4.1 CMAC的安全性 ……206
6.4.2 CMAC的设计 ……207
6.5 散列消息认证码 ……215
6.5.1 HMAC的设计 ……216
6.5.2 HMAC的实现 ……217
6.6 总结 ……222
6.6.1 MAC函数可以做哪些事 ……222
6.6.2 MAC函数不能用来做哪些事 ……224
6.6.3 CMAC与HMAC ……224
6.6.4 重放保护 ……225
6.6.5 先加密再MAC ……226
6.6.6 加密和认证 ……227
6.7 常见问题 ……236
第7章 加密和认证模式 ……239
7.1 简介 ……239
7.1.1 加密和认证模式 ……239
7.1.2 安全目标 ……240
7.1.3 标准 ……240
7.2 设计与实现 ……240
7.2.1 额外的认证数据 ……240
7.2.2 GCM的设计 ……241
7.2.3 GCM的实现 ……244

7.2.4 GCM的优化 ……262
7.2.5 CCM的设计 ……264
7.2.6 CCM的实现 ……265
7.3 总结 ……274
7.3.1 这些模式可以用来做哪些事 ……275
7.3.2 选择一个Nonce ……275
7.3.3 额外的认证数据 ……276
7.3.4 MAC标记数据 ……276
7.3.5 构造举例 ……277
7.4 常见问题 ……281
第8章 大整数算术 ……283
8.1 简介 ……283
8.2 什么是BigNum ……283
8.3 算法 ……284
8.3.1 表示 ……284
8.3.2 乘法 ……285
8.3.3 平方 ……293
8.3.4 Montgomery约简 ……299
8.4 总结 ……303
8.4.1 核心算法 ……303
8.4.2 大小与速度 ……304
8.4.3 BigNum库的性能 ……304
8.4.4 TomsFastMath算法库 ……305
8.5 常见问题 ……306
第9章 公钥算法 ……307
9.1 简介 ……307
9.2 公钥密码的目标 ……308
9.2.1 保密性 ……308
9.2.2 不可否认和真实性 ……308
9.3 RSA公钥密码 ……309
9.3.1 RSA简述 ……309
9.3.2 PKCS #1 ……310
9.3.3 RSA的安全 ……314
9.3.4 RSA参考资料 ……315
9.4 椭圆曲线密码学 ……316
9.4.1 什么是椭圆曲线 ……316
9.4.2 椭圆曲线代数 ……317
9.4.3 椭圆曲线加密系统 ……318
9.4.4 椭圆曲线的性能 ……323
9.5 总结 ……324
9.5.1 ECC与RSA ……324
9.5.2 标准 ……326
9.5.3 参考资料 ……326
9.6 常见问题 ……327

第1章

概 述

本章解决方案：

- 威胁模型
- 什么是密码学
- 资产管理
- 常识
- 开发工具

☑ 总结

☑ 快速查找解决方案

☑ 常见问题

1.1 简介

在日常生活中，计算机安全是一个很重要的研究领域。当我们打开手机、查看语音和电子邮件、使用付款机或信用卡、购买按次观赏计费的电影、借助EZ-Pass使用无线电发射机应答器、登录在线视频游戏甚至是在看医生的时候，就需要考虑计算机安全问题了。虚拟专用网（Virtual private Networks VPN）和安全Shell（Secure Shell Cohnection, SSH）连接可以允许员工远程访问计算机，这两者的建立也需要考虑计算机安全问题。

人们使用（通常都是没有正确的使用）密码学技术来解决安全问题都是基于一个原因：安全需求。但是安全需求往往开始并不明确，而通常是由于人们在接受了事实的教训之后或者更重要的是在系统被攻击之后才会明确自己对安全的需求。

来自安全研究人员的忠告

已知的系统攻击——Dark Age of Camelot

网址：http://capnbry.net/daoc/advisory20040323/daoc-advisory2.html

在2004年3月，有人制造了一个攻击程序，它利用视频游戏*Dark Age of Camelot*（Mythic娱乐公司）中用于安全金钱交易的服务器认证的弱点。这个攻击程序可以让攻击者截获真实的服务器和客户端之间的通信并且读取所有的个人金钱数据。

即使软件开发人员使用了公开的和已经测试过的加密算法库来实现核心加密算法，但是他们却没有正确地去使用。结果导致攻击者并不需要去破解像RSA和RC4这样复杂的密码学算法，只要利用那些漏洞就可以了。

Mythic攻击程序是一个可以说明软件开发者不知道怎样合理使用工具的经典例子。这很难去责备游戏的开发团队，毕竟他们是视频游戏开发人员而并不是专业加密人员。他们没有资源

去引进密码专业人员，只好和其他独立的公司签定合同来提供这些加密服务。

当时，几乎全世界的软件开发公司都像Mythic公司一样碰到了这种软件安全问题。当越来越多的小型企业不断成立时，他们用于解决安全问题的资源却越来越少。虽然安全并不是最终用户产品的一个目标，但它却是软件产品更加有用的一个必要条件。

例如，在银行业务中几乎不需要使用密码学，你根本不需要先进行一次RSA密钥交换就可以给别人10美元。同样，手机也不需要。在数字语音技术概念中，需要对使用无线电传输的数据进行压缩（解压缩）和编码（解码），而这也同样没有用到密码学技术。

由于安全并不是核心产品价值，所以它常被忽视或者降低到次要的“需求”目标列表中。更让人遗憾的是，密码学及其应用通常是一项非常容易的任务，并不需要有密码学或者数学高级学位的人来完成。实际上，大部分的安全任务只需要开发人员知道如何使用现有的密码工具就可以了。

1.2 威胁模型

威胁模型（threat model）明确地指出并研究系统中哪些地方可能会被攻击者利用来攻陷系统。如果是银行，攻击者就想要有效证件；如果是电子邮件服务，攻击者则需要私人消息，等等。简单地说，威胁模型就是考虑整个系统的一般应用过程来检测在极端情况（corner case）下会发生什么。也就是说，假如你期望得到的映射应该是在集合X中的，但是如果攻击者发送过来的不属于集合X，而属于另外一个集合Y时，会产生什么样的结果呢？

这种模型的一个最简单的例子就是C语言中的一个函数atoi( )，在实际编程应用中，人们通常并没有对其做错误检查。这个函数所期望的输入是一个经过ASCII编码的整数，但是如果输入的不是一个整数会怎样呢？虽然这几乎不是一个安全缺陷，但它的确是一种会被攻击者利用来攻击你的系统的一种极端情形。

威胁模型存在于包括内部人员在内的任何人都可以和系统交互的层次上。内部人员通常被看作是比较特殊的用户，他们可以不用受任何限制地访问系统数据，而他们常常又犯有把存储了成千上万个客户机密数据的笔记本电脑随手放在车里这种愚蠢的错误（例如：http://business.timesonline.co.uk/article/0,13129-2100897,00.html）。

系统被攻破以后攻击者可能会做什么，这种模型从根本上表示了系统在实际使用中将会碰到的用例（Use Cases）。例如，当用户首先需要提供一个密码时，攻击者会检查在密码的处理过程中是否存在弱点。同时，攻击者也会看看系统是否采用了防止猜测简单密码的安全策略，等等。

开发一种精确的威胁模型的最主要的影响因素是，不要以一种正常用户观点来考虑这些用例。例如，当你的程序向数据库提交数据时，攻击者可能通过在程序发送的数据中加入查询语句来实施注入攻击，而一名正常的普通用户可能永远都不会这么做。描述一种威胁模型设计的全过程几乎是不可能的，因为安全模型至少和系统自身的设计同样复杂。

本书并不打算提供一种安全的编程实践解决方案来解决这种问题。但是，威胁模型设计人员在设计系统时仍然需要考虑以下几种简单的规则。

威胁模型设计的几种简单规则：

1. 哪些情况会导致这种用例的出现?
   - 考虑预计之外的情况。
   - 无效的情况都处理了吗?
2. 输入交互还需要哪些处理模块?
   - “无效输入”会是什么样的?
3. 这种用例有效吗?
   - 是否有很明显的漏洞? 在假定的条件之下会出现什么情况?
   - 它是否实现预期目标?

1.3 什么是密码学

密码学是一种实现安全目标自动化的（或者算法形式的）方法。通常，我们所说的“加密算法”只是从在计算机上执行的角度来讨论一个算法，这些算法处理以位形式表示的信息。

更特别地是，人们通常认为密码学就是研究密码的，也就是隐藏信息的那些算法。实际上，与这种说法相符的真实名称叫“保密性”，而它仅仅是密码学研究领域中的一个方向而已。这之所以是最流行的对密码学的看法，大概因为它是传统密码需要实现一个安全目标并且也符合人们希望保守秘密的天性。以欲望、需要、错误和恐惧形式存在的秘密是每个人都有的自然情感。它当然也帮助了好莱坞拍摄像Swordfish（《剑鱼行动》）和Mercury Rising（《水银蒸发令》），又译作《终极密码战》这样的电影。

1.3.1 密码学的目标

虽然密码学是为了确保安全性的，但是它自身也存在着一些根本问题需要解决，当然它的重要性取决于谁来攻击你以及你为了对抗攻击者采用了什么样的安全措施。本小节中提到的密码学的目标包括：保密性、完整性、认证和不可否认（以讲述的先后顺序）。

1. 保密性

保密性是具有隐藏消息的真实含义和目的的属性。特别地，当消息在信息传输媒介中传输的时候，为了防止消息被那些不希望其了解消息真实内容的人所窃取，而把消息隐藏起来。这些传输媒介包括Internet、无线网络、蜂窝电话网络，等等。

典型的实现保密性的方法是使用对称加密算法。这些算法利用密钥对原始消息（即明文）进行加密产生几块信息（即密文）。从信息论的基本观点来看，密文和明文具有相同数量的信息熵。这也就是说密文的接收者只需要同样的密钥以及密文就可以重新还原出明文。

对称密码算法可以通过两种形式来举例说明，每一种都有自己要强调的地方，同时也都各有不足。本书只深入讨论分组密码，主要是美国国家标准与技术协会（NIST）的高级加密标准（AES）分组密码算法。AES分组密码非常流行，因为它无论是在大的还是小的处理器上都可以非常有效地运行并且在硬件实现上采用了低成本的设计技术。分组密码比它的“表兄”流密码要流行，因为它容易理解并且应用广泛。正如我们所见，AES可以用来创建各种保密算法（包括一种看起来像流密码的模式）以及完整性检验和安全认证算法。AES是免费的，并且从知识产权（IP）角度来看，它的说明文档很详细而且是基于可靠的密码学理论的（如图1-1所示）。

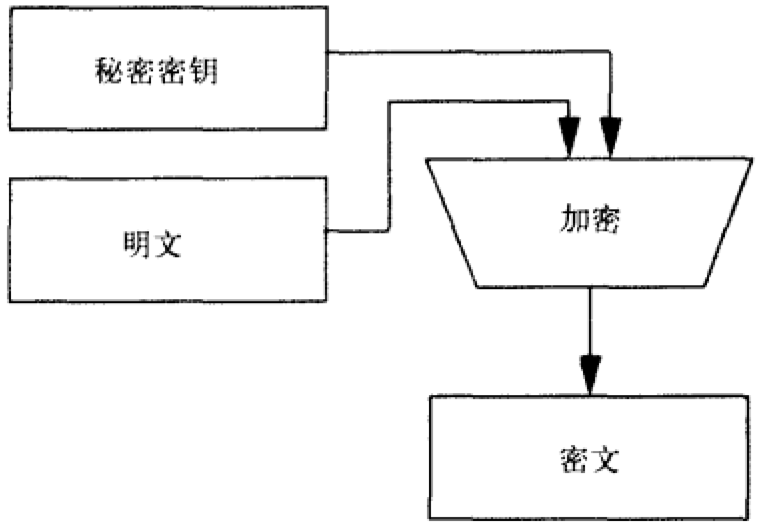


图1-1 分组密码算法的示意图

注意 网址: http://csrc.nist.gov/Crypto Toolkit/aes/rijndael/。高级加密标准（AES）是NIST官方推荐使用的分组加密算法。由于它的设计目的是用来替代使用已久并且速度很慢的DES加密算法，所以它目前在许多领域里有着非常广泛的应用。

2. 完整性

完整性是指在不存在一个活动的攻击者参与的情况下确保信息和数据的正确性。这听起来比其实际情况要复杂得多。简单地说，就是保证信息在从A传输到B的过程中，其含义（或内容）没有被修改。完整性仅限于攻击者并不打算篡改传输数据的正确性的情况下。

完整性检验通常是通过密码学中的单向散列函数来实现的。这些算法把输入看成可以是任意长度的消息并且产生一个固定大小的消息摘要。消息摘要（或者简称摘要）通常是在160到512位之间，它是消息的表示形式。也就是说，给定消息和与其相对应的摘要，可以判断消息在传输过程中有没有被修改过。散列函数的设计具有一些很有趣的特点，比如说要必须具备单向性并且能够避免冲突（如图1-2所示）。

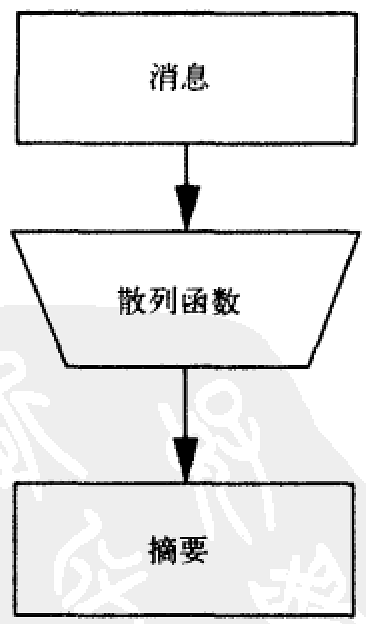


图1-2 单向散列函数的分块示意图

散列算法设计成单向的主要是，它是一种用来实现基于密码的认证系统的方法。这也就意味着如果给定一个消息摘要，人们在可行的时间内（低于指数级的）是不能计算出生成这个摘要的输入的。同样地，散列算法的单向性也是实现其他算法安全性的一个必要条件，例如第5章中的散列消息认证码（HMAC）。

散列算法也要求在两种情况下是可以避免碰撞的。一种情况是针对固定的消息要求避免产生预映射约束（pre-image resistant）（如图1-3所示）。也就是说，给定一个值*y*很难找到消息*M*使得*hash*(*M*)=*y*。第二种情况通常称为第二预映射约束（如图1-4所示），是指不能找到两个消息*M1*（给定的）和*M2*（随机选取）使得*hash*(*M*1) = *hash*(*M*2)。只有这两个条件都满足才能叫做碰撞约束。

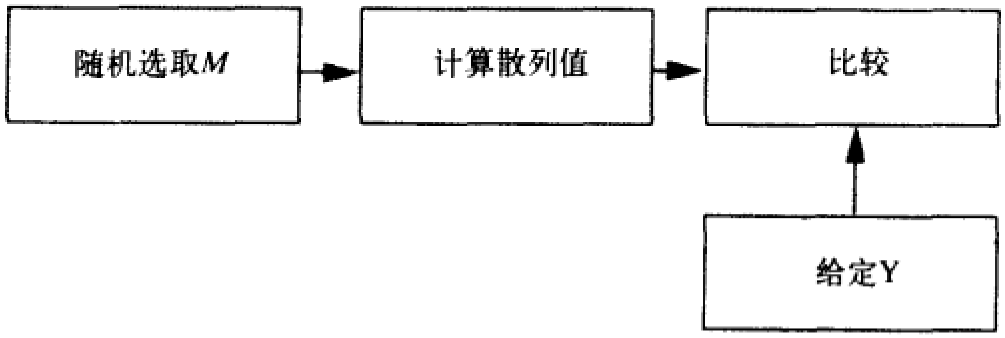


图1-3　预映射碰撞约束

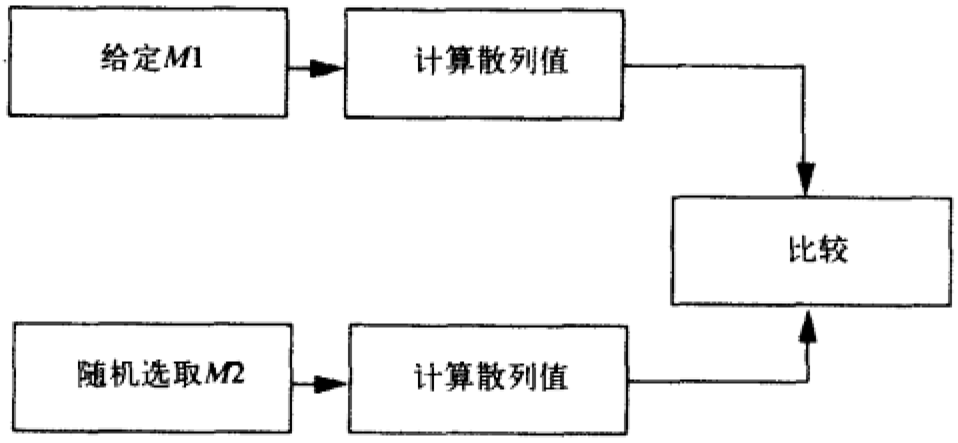


图1-4　第二预映射碰撞约束

散列并不是基于密钥的算法，也就是说在算法的整个流程处理过程中，没有什么不能让攻击者知道的机密信息。加密者可以计算一个公开消息的消息摘要，攻击者也可以。由于这个原因，在攻击者实施攻击的情况下，我们无法确定消息是否完整。

即使考虑到具有这种潜在的危险，但是散列算法还是被广泛地应用于计算领域。例如，大多数在线发布的Linux和BSD程序都提供了如md5sum之类的计算文件消息摘要的程序。一般情况下，作为用户更新过程的一部分，用户既需要下载文件也需要更新文件的摘要。假如威胁模型只考虑到像文件的重复和覆盖等存储以及发布上的错误，而没有考虑到攻击者因素，这种做法就很有用。

本书讨论了流行的安全散列标准（SHS）SHA-1和SHA-2系列散列算法，同时这也是NIST所设计的密码学算法中的一种。特别是SHA-2系列相当地具有吸引力，因为它引入了可以产生224到512位消息摘要的散列函数。考虑到这些算法不需要存储表和复杂的指令，所以它们的效率很高，而且也可以根据指定的信息重新产生消息摘要。

警告！ MD5散列算法的安全性已经变得很弱。Dobbertin发现了该算法关键部分的缺陷，并且在2005年研究人员发现了该算法中函数的完全碰撞。2006年初发表的一些论文讨论了如何更快地寻找碰撞的方法。

这些研究人员大部分是研究第二预映射碰撞，但已经存在利用这种碰撞攻击IDS和分布式系统的方法。

强烈建议开发者避免使用MD5散列函数。在一定程度上，为了更好地使用新的SHA-2散列算法，甚至SHA-1散列算法也应该避免使用。对于下文中提到的欧洲标准，Whirlpool散列算法也可以作为一种选择。

3. 认证

认证是指身份特征或者消息实体的代表的属性。一个经典的例子就是用于信件上的蜡封。这个标记在使用的时候很难被伪造，并且完好无损的标记意味着文档是经过认证的。

另外一种认证的常见形式是输入个人识别码（PIN）或者密码来认证一次交易。不要和不可否认性混淆，不可否认性是为了防止别人否认协议的；同样也不能和认证协议相关的密钥协商与建立协议相混淆。当我们说在认证一个消息时，意思是指：在敌人可能会对消息进行伪造的情况下，采取额外的步骤以便接收者能够对消息的完整性进行验证。

密钥协商的过程和可靠性的相关问题是公开密钥协议的一个研究方向。它们使用大致相同的基本操作，但却有着不同的约束和目标。一个认证算法通常意味着是对称算法，以便所有参与认证的成员都能生成可以验证的数据。具有不可否认这一性质的可靠性通常只有一个认证信息的生产者以及多个验证者。

在密码学世界中，这些认证算法通常叫做消息认证码（Message Authentication Code MAC），并且像散列函数一样产生一个固定大小的输出，称为消息标记。这个标记是验证者用于检验文档的信息。和散列函数不同的是，这一系列的MAC函数需要秘密密钥来阻止任何人对标记进行伪造（如图1-5所示）。

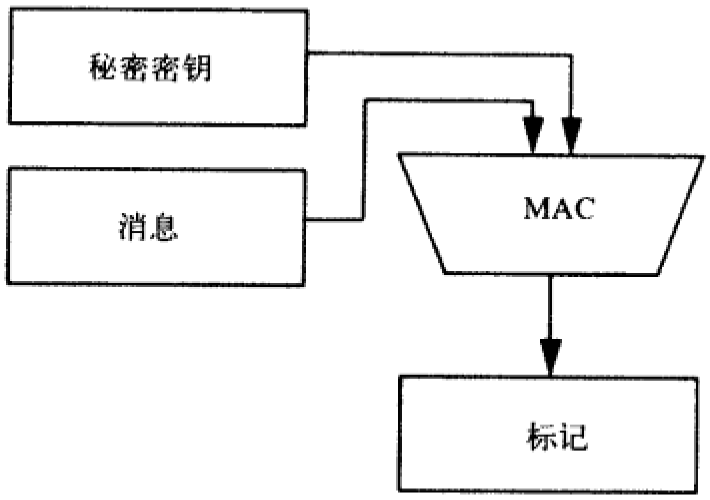


图1-5 MAC函数的分块示意图

两种最常见的MAC算法形式是CBC-MAC（现在由OMAC1算法实现并且在NIST圈内称为CMAC）和HMAC函数。CBC-MAC（或者叫CMAC）使用分组密码算法，而HMAC使用散列函数。本书包含了由NIST认可的CMAC以及HMAC消息认证码算法。

另外一种实现认证的方法是使用公钥算法，如使用PKCS#1标准的RSA或者椭圆曲线DSA（EC-DSA或者ASNI X9.62）标准。与CMAC或HAMC不同的是，基于公钥的认证算法不需要双方在通信之前共享私有信息。因此当处理随机的、未知的参与认证成员时，公钥算法并不局限于在线交易这一领域。也就是说，你可以对一个文档进行签名，那么任何一个拥有公钥的人都可以进行验证而事先并不需要和你联系。

公钥算法的使用方法不同于MAC算法，这些将以另外一个主题在后面的章节里讨论（如表1-1所示）。

表1-1 CMAC、HMAC和公钥算法的比较

特征	CMAC	HMAC	RSA PKCS#1	EC-DSA
速度	快	最快	最慢	慢
复杂性	简单	最简单	困难	最困难
需要秘密密钥	是	是	否	否
实现的简单性	困难	困难	简单	简单

使用什么样的方式认证取决于认证者是如何构造认证系统的。基于公钥的认证系统通常用

于一个新建立的交流媒介的初始认证。例如，当你第一次连接到一个使用SSL的网站时，你需要验证这个网站的证书签名确实是由合法签名者所签的。相反，当传输信道已经安全的情况下，才会使用像CMAC和HMAC这样的算法。使用它们是为了保证数据流在传输过程中没有被篡改。由于CMAC和HMAC处理数据的速度极快，所以它们在流量大的传输媒介中能发挥更高的价值。

4. 不可否认

不可否认具有同意负责任这一属性。更确切地说，是无法反驳应尽的责任。例如，你拿了一支笔在（具有法律效力的）合同上签了自己的名字，那么你的签名就是一种不可否认的"设备"。事后你不能不同意合同上的条款或者拒绝承认曾经同意了合同。

不可否认和认证的性质十分相似，因为从实现的角度上来讲，它们大都使用了相同的基本原理。例如，当且仅当一个特殊的成员具备签名能力的时候，他的公钥签名才能称为不可否认"设备"。因此，像CMAC和HACM这样的其他MAC算法都不能叫做不可否认设备。

不可否认在现金交易和审计工作中的作用非常重要，而这一点却常被忽视。例如，信用卡收据上面的钢笔签名很少有人去效验，即使职员看了一眼信用卡的背面，但由于他不是笔迹专家，所以也不能从实物中分辨出细微的伪造之处。手机也经常使用MAC算法作为资源使用认证者，因此它不具有不可否认性质。

5. 目标概括

表1-2比较了4种主要的密码学目标。

表1-2 加密的一般目标

目 标	性 质
保密性	1. 主要是为了防止那些不希望其知道信息内容的人从信息传输媒介获取信息而将信息隐藏起来 2. 多使用对称加密算法 3. 接收者并不知道信息是否完好无损 4. 加密的输出为密文
完整性	1. 主要用来保证信息在传输过程中的正确性 2. 设想没有敌人的干扰 3. 多使用单向散列函数 4. 散列函数的输出为消息摘要
认证	1. 主要用来保证信息在传输过程中的正确性 2. 设想存在敌人的干扰 3. 多使用消息认证函数（MAC） 4. MAC的输出为消息标记
不可否认	1. 主要用于约束交易行为 2. 目的是防止成员否认参与交易 3. 多使用公钥数字签名算法 4. 签名算法的输出为签名

1.4 资产管理

在实现密码学算法的时候，安全和有效地管理用户的资产和证书（credential）的能力是个很大的挑战。所谓的资产可以是任何一样事物，从信息、文件到医疗信息、联系人列表等，以

及用户所拥有的但不是用来鉴别用户的东西。相反地，证书是用来鉴别用户的。一般而言，证书是用户名、密码、PIN码、两因素认证令牌和RFID标签等信息。证书也包括用于RSA和ECC签名的私钥等信息。

资产一般并不以特殊的安全措施来管理。一般来说，它们假设是可以认证的并且很少对隐私进行保护。很少有程序为资产提供集成的安全特性，而是设想用户会自己添加额外的工具，这些工具有加密的文件存储器或者需要手动处理的GnuPG等。在实际应用中，资产常被误认为是证书。

例如，在2005年，飞往加拿大仅需要一张信用卡。当在加拿大国内旅游时，检票终端会自动检索电子票据，寄宿代理商并不检查身份证。这些系统设想拥有信用卡就意味着买票的和站在检票厅的是同一人。

1.4.1 保密性和认证

与资产相关的两个重要的问题是资产是否是私有的以及资产是否是完好无损。例如，许多磁盘加密用户使用工具给整个系统驱动器进行加密。许多系统文件通常作为软件安装媒介的一部分。它们并不是私有资产，对它们进行加密实在是浪费资源。更糟糕地是，大部分的加密系统都很少针对磁盘上的文件使用认证工具（EncFS是极少的一个例外。http:// encfs.sourceforge.net/），这意味着用户访问的大部分文件包括应用程序都可以被修改。通常，它们可以对系统发动拒绝服务攻击，但这完全可以用改变系统初始流程的方式来修改一个正在运行的程序。通过使用认证方案，可以限制系统文件为可读或者不可读。任何修改都将被简单地禁止。

一般而言，对那些不用于公开的信息进行认证也许是种不错的方法。这给用户提供了两种形式来保护他们的资产。考虑到这是一种新兴的趋势，NIST和IEEE都已经推荐使用混合加密模式（分别是CCM和GCM），这种模式确实提供了加密和认证。这些模式本身也是理论上所关心的问题，因为在形式上归结为它们的继承原语的安全性（通常是AES分组密码）。从性能角度来看，这些模式比单独的加密或者认证效率更低。但是，它们提供给用户的是稳定性。

1.4.2 数据的生命周期

在某种资产和证书的整个生命周期过程中，需要更新或者删除部分资产。不像简单地创建一种资产那样，对资产做进一步的改变所带来的安全问题并不是微不足道的。如果一些操作模式的参数是静态的，那么它是不安全的。例如，CTR链接模式（将在第4章中进行讨论）在加密数据时需要一个初始值（IV）。简单地重复使用一个初始值，也就是说允许内联修改，这在抵抗保密性威胁时是一点都不安全的。类似地，像CCM（参见第7章）等其他的模式，对每组消息需要一个新的随机值Nonce用以确保输出的隐私性和可认证性。

某些数据，例如医疗数据，也包含平均生命周期，用密码学的术语来讲叫做访问控制限制。这通常使用指定了过期日期的公钥数字签名技术来实现。严格来讲，访问控制需要一个可信任的发布成员（例如一台服务器），它遵守针对被发布的数据制定的规则（例如严格自律）。

有序数据流的概念产生于基于状态的通信需求。例如，当你向一台服务器发送银行业务请

求时，你希望请求仅被允许和接受一次，尤其是在结账的时候。当攻击者能够在一次被接收方接受的通信会话中重新激活某个事件的时候，就会产生重放攻击。针对这个问题的最简单（初始的）的解决办法是在认证消息中引入时间戳或者递增计数器。

在密码学和通俗计算机科学中，时间戳概念也是一个令人头疼的问题。对于始发者，时间是多少？我的电脑的时间好像和你的并不一样（当以秒来衡量时）。当我们使用没有经过校时并且可能不是同步改变的时钟进行在线实时通信时，这种情况会变得更加糟糕。当许多在线协议和无序的UDP网络结合时，它们将很快变得令人厌烦。由于这些原因，相对于计时器来说，更加需要计数器。但是，不像听起来那么好，计数器并不实用。例如，离线协议没有计数器的概念，因为它们所看到的消息往往是它们看到的第一个消息。并没有“第二个”消息。

简单地在数据信道的认证部分中包含计数器是个不错的准则，更重要的是，要检查这个计数器。有些时候，安静地拒绝无序的消息比不管消息的顺序就盲目地允许所有的传输要好得多。对于GCM和CCM模式，使用初始值IV（或者依赖Nonce）和即用初始值IV也用计数器，这两者差不多。

提示　对于像GCM和CCM这样的协议中的初始值IV或者Nonce有一个简单的小技巧，就是使用少数字节作为包计数器。例如，当数据包少于65 536个明文字节时，CCM模式可以接受13个字节的Nonce。如果你将有小于4亿的数据包需要发送，那么就可以用前4个字节作为包计数器。更多的信息请参考第7章。

1.5　常识

在密码研究者中间有这样一个常识，安全最好留给密码学家；密码学算法的讨论会导致人们很快地误用这些算法。一个典型的例子如下，Bruce Schneier 在他的Secret and Lies（《秘密与谎言》）的摘要中写道：

我写这本书一方面是为了纠正一个错误。

几年前，我写了另外一本书：*Applied Cryptography*（《应用密码学：协议、算法与C源程序》）⊖。其中，我描述了一种数学的完美境界：算法可以保护你的最深的秘密数千年，协议可以使最理想的电子交易活动——无节制的赌博，无法觉察的认证，匿名的现金交易——安全和万无一失。以我之见，密码学是最伟大的技术平衡器。任何人只要拥有便宜（并且每年越来越便宜）的计算机都可以和大的政府一样拥有相同的安全保障。在两年之后所写的这本书的第二版中，我更进一步地写道：用法律来保护我们自己还不够；我们还需要数学来保护我们自己。

摘要Secret and Lies, Bruce Schneier。

在这个引用中，他简要的讲述了这样一个概念，从总体上来说，密码学本身并不能让我们“安全”。在实际生活中，这基本上是正确的。这有许多活生生的例子，本来是很完美的密码算法验证体系，但是因为用户的错误使用或者物理攻击（例如信用卡窃取）而导致被破解。但是，向外行传播密码学知识这一想法多多少少会让人感到惭愧，而且不可避免地起不到太多的作用。

⊖　《应用密码学：协议、算法与C源程序》一书已由机械工业出版社于2000年1月出版。

依赖于安全专家似乎是一种能够获得安全的系统的有效办法，但这完全不实际，也没有任何作用。本书一个主要的体系和目标是为了打消人们这样一种想法：密码学比实际情况复杂得多（或者密码学需要比实际情况更加复杂）。通常，对威胁的清醒认识会帮助你针对安全需求简单并且高效地设计加密系统。现在已经有许多看起来安全的产品出现，开发人员只要对这些产品进行足够的研究，找出适合自己任务的工具和安全地使用它的方法即可，这些工具包括软件库、算法或者各种各样的实现。

请记住本书要说明的是什么。我们阐述了密码学家设计的标准算法，并指出怎样合理地使用它们。这是两项不同的任务。首先，这需要很强的密码学和数学背景知识。其次，我们需要理解的是这些算法将解决什么问题，它们为什么是这样设计的以及怎样使用它们。

最近，我们访问了一个视频游戏开发小组并且检查了他们的加密系统，这个加密系统工作于幕后而且远离最终用户。我们发现了一些无关紧要的地方，这些地方每个人都会有不同的设计。但是，在他们开始开发工作几个小时后，我们真的没有发现任何能够一眼就瞧出来的错误。他们很明显地考虑到了威胁模型，而且在他们有限的计算能力条件下，也针对模型设计出了可以工作的系统。简单地说，他们设计出了一种可以工作的系统，这个系统允许他们的产品能够在自然条件下有效地工作。在这个开发小组中没有一个人研究过密码学，也没有一个人是密码学的业余爱好者。

仅仅简单地知道你需要安全，也知道存在相应的算法，这会导致具有致命不足的结果。部分来看，这也正是本书之所以存在的原因：说明存在哪些算法，怎样以不同的方式来实现它们，配置和使用它们的时候要注意什么。这是一本即讲解了组成又指出了操作指令的书籍。

1.6 开发工具

纵观本书，我们使用了一些很容易获得的工具，例如适合X86 32位和64位平台以及ARMv4处理器的GNU Compiler Collection C 编译器（GCC）。之所以选择它们，是因为它们非常专业，可以免费使用并且对读者具有真正的价值。

书中出现的各种算法大部分是以可移植的C语言实现，尽可能地方便更多的读者来阅读下面的那些代码列表。这些代码列表都是读者可以马上拿来就用的C代码，同时读者也可以在相应的网站上获得这些代码。同样，对C编译器产生的汇编代码列表也进行了适当的分析。如果读者熟悉AT&T风格的X86汇编语言和ARM风格的ARMv4汇编语言就更好不过了。

对于大部分的算法和问题，可能都有多种实现方法。例如，乘法就有3种基本的算法，并且每种算法又有各种不同的实现技术。对各种配置的大小和效率在列出的平台上也进行了一些适当的比较。当然，也鼓励读者在书中没有列出的平台上对这些配置进行测试比较。

一些算法也有安全权衡问题。例如，AES在使用查找表时速度最快。在这种配置条件下，它也有可能泄漏侧信道（side channel）信息（在第4章进行讨论）。本书探讨了各种变化试图最小化这种泄漏情况的发生。对这些算法实现的分析紧紧联系了现代处理器的设计技术，尤其是具有数据缓存功能的。建议读者能够至少熟悉X86处理器（例如Intel Pentium 4和AMD Athlon64）是怎样在块层次上进行操作的。

有时在书中会参考一些现有的源代码，例如TomsFastMath和LibTomCrypt。这些都是用C

编写的开源算法库，并且已经用于工业生产中的各种平台，小到传感器，大到企业服务器。而本书中的代码列表是不依赖于这些算法库的，建议读者参阅算法库，研究它们的设计，甚至也可以使用它们来做“重复造轮子”之类的事情。这并不是说你不需要自己试着实现这些算法；相反地，在环境允许的条件下，使用发布的源代码可以缩短产品开发和测试周期。当这些算法库不适合项目的限制时，建议读者使用自己的实现。这些项目都可以在www.libtomcrypt.com网站上获得。

在X86处理器上的时间数据是通过RDTSC指令生成的，它提供了精确的周期时间数据。建议读者熟悉这条指令及其用法。

1.7 总结

我们将要学习的专业加密系统的开发并不是一个非常复杂和高层次的实践。通过明确地定义包含系统暴露出的弱点的威胁模型，人们就可以开始理解密码学在产品的安全中发挥什么样的作用了。当一个人不明白怎样去发现脆弱点的时候，密码学就成为惟一难以解决的问题了。

可以从用例的所有使用情况来创建一个威胁模型。当能够用“是”或者“没有应用”来回答“这个模型有没有解决保密性？”以及“这个模型有没有针对可认证性问题？”这两个问题的时候，这个模型已经创建完成了，至少完成了一部分。

知道什么地方发生错误是创建一个安全的加密系统的第一步。紧接着，需要决定为了解决问题要采用什么样的加密方式，是用分组算法、散列、随机数生成器、公钥算法、消息认证码还是挑战－应答系统。如果你设计的系统需要一个RSA公钥，而用户却没有，显然这个系统是不行的。证书和资产管理是加密系统设计所不可或缺的组成部分。它们决定了什么样的加密方式是适合的，以及你必须保护什么。

产品运行环境的约束决定了程序的代码和数据驻留的可用空间（内存），以及给定任务的处理能力。这些约束时常对算法的选择以及后面的协议设计起着关键的作用。例如，在受到CPU限制的时候，ECC可能比RSA更加适合，然而在某些情况下，不用公钥算法更加实际。

从程序的自身属性到用户需要访问的证书和资产再到运行时环境，所有的这些设计因素都应当在设计加密系统时综合考虑。这种综合和设计合适的解决方案是安全工程师应该做的。本书通过讲述什么样的，如何使用以及为什么使用密码学算法，来指出安全工程序师需要什么。

1.8 本书的组织结构

本书以章节的形式根据问题的种类进行组织。按照这种方式，每一章都完整地讲解了密码学以及开发者所关心的各个领域的问题。本章从一般意义上对密码学的主题做了一个简单的介绍。如果读者希望进一步了解密码学的各个方面，建议读者同时阅读其他书目，例如*Applied Cryptography, Practical Cryptography*或者*The handbook of Applied Cryptography*。

第2章“ASN.1编码”，讲述了用于数据元素的抽象语法记法一（Abstract Syntax Notation One，ASN.1）⊖，这些数据元素有串、二进制串、整数、日期和时间、集合和序列。ASN.1通

⊖ 也有译作抽象语法标记一，抽象语法符号一，本书以中国国家标准GB/T 16263.1-2006/ISO/IEC 8825-1:2002中的翻译为准。——译者注

常用于公钥标准中，它作为一种在多平台环境下传输多类型数据结构的标准方法。ASN.1也可用于通用数据结构，因为它是一个标准而且有一套很容易理解的编码规则集。例如，你可以用ASN.1格式对文件头进行编码，也可以赋予你的用户自已使用第三方工具对文件头进行调试和修改的能力。任何使用ASN.1编码而不使用自已建立的标准的工程，都会得到很大的"增值"。本章研究ASN.1规则的一个子集，它特用于常见的密码任务。

第3章"随机数生成"，讨论像由NIST指定的标准随机数生成器（RNG）的设计与结构。RNG和伪（或确定性的）随机数生成器（PRNG或者DRNG）是所有加密系统中很重要的一部分，因为许多算法是随机的而且需要不可预测且不重复的材料（material）（例如初始值或nonce）。由于PRNG形成了所有加密系统的实质，所以从逻辑上应该从它们开始密码学的讨论。本章探讨了各种PRNG的结构，怎样对它们进行初始化、维护以及要避免哪些有潜在危险的地方。建议读者在自己的实际设计中也采用同样的原理。总是先处理它们的"随机位"出现的地方是很重要的。

第4章"高级加密标准"，讨论AES分组密码的设计、实现上的折衷、侧信道危险以及作用的模式。这一章只粗略地提供了AES的设计，更多地关注对实现者来说是关键的设计元素以及怎样在各种权衡条件下使用它们。Bernstein的数据缓存侧信道攻击在这里也做了描述，主要是作为一种设计上潜在的危险来讲的。这一章也给出了用于AES分组密码的CBC和CTR模式，特别关注这些模式所用于解决的问题，怎样对它们进行初始化，以及各自使用上的危险。

第5章"散列函数"，讨论NIST的SHA-1和SHA-2系列单向散列函数。首先描述了它们的设计，然后是实现上的一些折衷。这一章也讨论碰撞约束，提供了利用的例子以及不正确地用法类型。

第6章"消息认证码算法"，讨论HMAC和CMAC消息认证码算法，它们分别是由散列和分组函数所构造的。按顺序对各种模式进行介绍，描述了其设计、实现上的折衷、目标和用法上的危险。特别关注使用计数器和计时器来解决在线和离线环境中的防止重放攻击的问题。

第7章"加密和认证模式"，分别讨论IEEE和NIST的加密和认证模式GCM和CCM。这两种模式都给密码学函数引入了新的概念，这一章就从那儿开始。特别地，它引入了"额外的认证数据"的概念，它是待认证但不用加密的消息。这给密码学算法的使用又增加了一个新的方向。GCM和CCM的设计按顺序进行介绍。特别研究了GCM的基本数学元素所拥有的表驱动实现。类似于MAC那一章，这一章所关注的仍然是重放攻击的概念，而且更深层次地探讨了初始化技术。GCM和LRW模式是相关的，因为它们共享一种特殊的乘法运算。建议读者在阅读这一章之前，首先阅读第4章的LRW模式。

第8章"大整数算术"，讨论用于操作大整数的技术，例如那些用于公钥算法中的。它着重介绍了开发者在程序任务中将面临的瓶颈算法。建议读者阅读可以从网站上获得的补充资料《Multi-Precision Math》以获得进一步的认识。这一章主要关注了快速乘法、平方、约简和幂运算，以及各种实际折衷。代码大小和性能的比较是这一章的亮点，它给读者提供了代码设计上的一种有效的导引。这一章稍微提到了数论中的各种话题，它们足够用来有效地阅读第9章。

第9章"公钥算法"，引入公钥密码学。首先介绍的是RSA算法和与它相关的PKCS #1填充机制。这一章给读者介绍了各种时间攻击以及它们各自的对抗措施。接着讨论的是ECC公钥算

法EC-DH和EC-DSA。在探讨ANSI X9.62和X9.63标准时，它们将使用NIST的椭圆曲线。这一章以各种椭圆曲线点乘函数的形式引入了新的数学，每一个函数都有各自性能上的折衷。建议读者阅读《Guide to Elliptic Curve Cryptography》一书以对椭圆曲线数学背景知识进行更深的了解。

1.9 常见问题

下面的常见问题，由本书的作者所回答，它们既可以用来测试你对本章所出现的概念的理解，也可以帮助你在实践中实现这些概念。如果希望作者解答你的问题，请浏览www.syngress.com/solutions，然后点击“Ask the Author”表单。

问：威胁模型开发应该在什么时候开始？

答：通常是和项目自身的设计同时进行。但是，考虑到像运行环境约束可能无法提前知道的这种情况，这会影响到威胁模型解决方案的精确度。

问：使用我们自己的加密算法来代替像AES和SHA-2这样的算法有没有什么好处？

答：通常没有，因为这关系到安全问题。针对具体的平台设计一种更加有效的算法是完全可行的。但是应该避免这种设计想法，因为它们将产品从遵守标准的领域中删除，而转移到了怀疑论者范畴。不严格地说，使用最佳实践，包括使用标准算法可以让你承担有限的责任。

问：什么是认证了的密码？什么是FIPS认证？

答：FIPS认证（参见http://csrc.nist.gov/cryptval/）是对指定算法的二进制或者物理实现通过FIPS授权检验中心认证的过程，结果或者是通过（并且经过认证）或者是实现在某些特定条件下失败。不能检验一种设计或者源代码。取决于产品的要求，认证分为各种级别：一级是一系列的测试，四级是对侧信道的物理审计。

问：在哪里可以找到LibTomCrypt和TomsFastMath项目？它们工作于什么平台？它们以什么样的方式授权？

答：它们在www.libtomcrypt.com。它们使用MSVC和GCC进行编译，支持所有的32位和64位平台。两个项目都是公开的并且可以免费用于各种目的。

PDG

第2章

ASN.1编码

本章解决方案：

- ASN.1概述
- ASN.1语法
- ASN.1数据类型
- 实现
- 总结

☑ 总结
☑ 快速查找解决方案
☑ 常见问题

2.1 ASN.1概述

抽象语法记法一（ASN.1）是ITU-T的一个标准集，它用来编码及表示通用数据类型，这些数据类型有可打印串值、八位位组串值、位串值、整数值以及用可移植方式组合而成的其他类型序列值。简言之，ASN.1指定了以何种方式对非平凡的数据类型进行编码，以便其他任何平台及第三方工具都能够解释其内容。

例如，在某种平台上，字母“a”可能以ASCII（或者IA5）编码为十进制数值97，而在其他的非ASCII平台上，可能会有另外的编码。ASN.1指定了一种编码方式，在任何平台上都可能对字符串进行统一的解码。这种标准同样给大整数、位串值和日期等平台敏感的数据类型带来编码和解码上的好处。

这对于开发者和客户端（或者消费者）都有同样的好处，因为它能够让发送的数据具有良好的模块化和可解释特性。例如，开发者可以告诉客户，他们所购买的程序数据是一种其他开发者也可以进行维护的格式，这样避免了制造商绑定问题，而且建立了一种互信关系。

确切地说，我们所关心的ASN.1规范是ITU-T X.680规定的，它描述了我们将在密码学应用程序中碰到的通用数据类型。在密码学中之所以使用ASN.1，是因为它把各种本身就不是可移植的数据类型真正地用字节来进行编码，并且正如我们将要看到的，是以一种确定的形式进行编码。对于那些为了避免歧义而要求对任一给定消息都有惟一编码的签名协议来说，确定的形式显得尤为重要。

ASN.1支持基本编码规则BER（Basic Encoding Rules），正则编码规则CER（Canonical Encoding Rules）和非典型编码规则DER（Distinguished Encoding Rules）（如图2-1所示）。这3种模式指定了怎样对相同的ASN.1类型进行编码和解码。它们的不同之处在于所选择的编码规则。我们将看到，特别是每种参数都有多种不同的编码，例如域值，布尔表示。DER和CER两

者都制定了完整的确定编码规则。也就是说，多种编码规则实际上可能有效的只有一种。

基本编码规则是最简单的编码规则集，它对于相同的数据可以有各种不同的编码。实际上，任何ASN.1编码（CER或者DER）都可以解码成BER，但是反过来不成立。所有ASN.1允许的数据类型首先都要用BER规则描述，然后才可以应用于CER或者DER。实际的完整ASN.1规范远比一般的密码学应用要求复杂得多。我们仅讨论DER规则，因为它是大多数密码学标准所要求的规则。要注意的是即使在开始时会讨论一些“结构化”编码，但我们并不支持它。

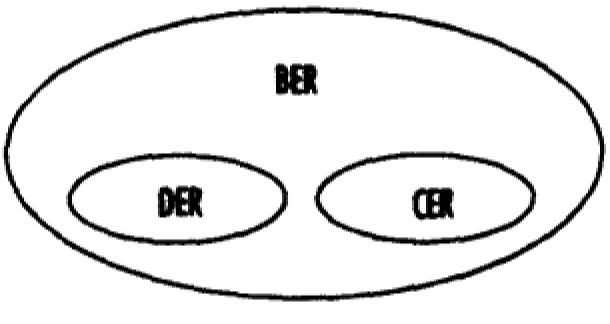


图2-1 ASN.1编码规则集

ASN.1最初于1994年标准化，并先后在1997年和2002年进行修改。当前的标准是ITU-T X.680，它也是ISO/IEC 8824-1:2002的文档，可以在网上免费获得（http://asn1.elibel.tm.fr/standards/）。本章讨论的是ASN.1编码的实现，而不是ASN.1的设计理论。作为进一步的了解，建议读者阅读ITU-T文档。我们学习ASN.1的目的是为了理解PKCS#1/#7和ASNI X9.62/X9.63这两个公钥密码算法标准。事实证明，为了支持这些标准，你的ASN.1过程代码不得不用DER规则集处理大量的ASN.1类型，同时，这些代码也适用于大范围的其他任务。

虽然正式的ASN.1不是仅为了密码学任务而设计的，但密码学项目也在相当程度上忽视了它。自定义的头部通常使用专用格式编码，这种编码并不是可通用解码的（例如一些ANSI X9.63数据），这造成要获得合适的并且可交互操作的软件远比需要的更加困难，所以导致采纳这种自定义编码作为标准的进度十分缓慢。当我们进行到本章最后时，在实际软件中使用ASN.1编码就会变得相当容易，而且它对于开发者和最终用户都是很方便的。

2.2 ASN.1语法

ASN.1语法遵循相当传统的巴科斯范式BNF（Backus-Naru Form）风格，但它在密码学领域中的介绍相当少。对我们而言，这个语法中重要的元素是名称、类型、修改器、允许值和容器。前面提到，我们只着重强调语法的语义。这部分的目标是让读者能足够地了解ASN.1的定义，以便于实现编码器和解码器来处理它们。最基本的表达式如下（在ITU-T文档中也做了定义）：

```
Name ::= type
```

按照字面意思表述就是定义某个名称为“Name”的元素，它是一个给定ASN.1类型“type”的实例。例如：

```
MyName ::= IA5String
```

意思是我们定义了一个名为“MyName”的元素或者变量，其类型为ASN.1类型IA5String（类似于ASCII字符串）。

2.2.1 ASN.1显式值

有些时候，我们需要定义一种ASN.1类型，它的子集元素包含预定义值。这可以通过下面的语法来完成。

PDG

```
Name ::= type (Explicit Value)
```

显式值（Explicit Value）必须是ASN.1类型允许选择的值，而且也必须是元素所允许的值。例如，使用前例我们可以指定一个默认的名称：

```
MyName ::= IA5String (Tom)
```

它的意思是说“MyName”是字符串“Tom”的IA5String编码。为了让这种语言具有更好的灵活性，这种语法还允许对显式值进行其他形式的声明。一个常见的例外是竖线（|）。再次展开前面的例子：

```
MyName ::= IA5String (Tom|Joe)
```

它的意思是字符串的值即可以是“Tom”也可以是“Joe”。这种语法的使用是为了扩展确定的解码器。例如：

```
PublicKey ::= SEQUENCE {
   KeyType      BOOLEAN(0),
   Modulus      INTEGER,
   PubExponent  INTEGER
}

PrivateKey ::= SEQUENCE {
   KeyType          BOOLEAN(1),
   Modulus          INTEGER,
   PubExponent      INTEGER,
   PrivateExponent  INTEGER
}
```

不要在刚刚使用的结构上停留太久。举这个例子的目的是为了说明通过一个显式定义的布尔值，就可以很容易地把两个相似的结构区分开。也就是说，当解码器在分析结构时，它首先遇到“KeyType”元素，然后就能决定怎样去分析余下的已编码的数据。

2.2.2 ASN.1容器

容器（container）是指一个包含了其他相同或者不同类型元素的数据类型（例如序列值（SEQUENCE）或者集合值（SET）类型）。这样做的目的是组合一些复杂的数据元素集，以便于在一个逻辑元素中进行编码和解码，甚至将其包含在更大的容器内。

ASN.1规范定义了4种容器类型：序列、单一序列（SEQUENCE OF）、集合和单一集合（SET OF）。虽然它们的意义不同，但是语法定义是一样的，如下所示：

```
Name ::= Container { Name Type [Name Type …] }
```

方括号中的内容和容器的元素个数都是可选项。我们将要看到某些容器允许在容器内部按类型进行分类。为了简化表达式以及方便阅读，常常是每行定义一个元素，并以逗号分开。

```
Name ::= Container {
    Name Type,
   [Name Type, …]
}
```

这和上例定义了相同的元素。容器的嵌套也是以同样的方式定义。

```
Name ::= Container {
   Name Container {
      Name Type,
      [Name Type, …]
   },
   [Name Type, …]
}
```

下面是一个完整的例子。

```
UserRecord ::= SEQUENCE {
   Name SEQUENCE {
       First  IA5String,
       Last   IA5String
   },
   DoB  UTCTIME
}
```

最后一个例子是粗略地将上例翻译成C语言中的结构来表示。

```
struct UserRecord {
   struct Name {
      char *First;
      char *Last;
   };
   time_t DoB;
}
```

我们将要看到，实际的容器解码函数并不像上例那样直接。但它们灵活性非常高，而且非常容易理解。

当然，在表达式中混合定义容器也是可行的。

2.2.3 ASN.1修改器

ASN.1定义了各种修改器，如可选（OPTIONAL）、默认（DEFAULT）和选择（CHOICE），它们可以改变表达式的声明。典型地用于定义一种要求编码灵活，而定义又不繁琐的类型。

1. 可选

可选，正如其名称所言，改变一个元素以便在编码时它的类型是可选择的。即编码器可以忽略这个元素，解码器不能假设它将出现。当邻接的两个元素具有相同的类型时，这会给解码器带来一些问题，也就是说为了合理地分析数据，解码器需要看前面的定义。基本的可选修改器定义如下所示：

```
Name ::= Type OPTIONAL
```

在容器中这样使用会存在一些问题，例如：

```
Float ::= SEQUENCE {
   Exponent     INTEGER OPTIONAL,
   Mantissa     INTEGER,
   Sign         BOOLEAN
}
```

当解码器读取这个结构时，在它看来第一个整数（INTEGER）可能是Exponent，也有可能

糟糕地认为是Mantissa。解码器必须再向前看一个元素才能决定这个容器的合理解码。

一般来说，并不建议使用这种方式定义结构。通常，有更简单的方法来表达一个容器，就是冗余一些，但这样解码器就可以在解码开始之前就决定该如何解码了。这样程序员也可以很容易地审计和复查。但是，我们将看到，可以使用一种灵活的链表解码模式对任意的ASN.1已编码数据进行一般解码。

2. 默认

默认修改器允许容器包含默认值。标准文档是这样定义的："集合值或者序列值的编码不应该包括任何等于它的默认值的组件值编码"（ITU-T建议书X.690国际标准8825-1（ITU-T Recommendation X.690 International Standards 8825-1）第11.5节）。它的意思很简单，是说如果待编码的数据值等同于它的默认值，那么它将在发送的数据流中被忽略。例如，考虑如下容器定义：

```
Command ::= SEQUENCE {
   Token  IA5STRING(NOP) DEFAULT,
   Parameter INTEGER
}
```

如果编码器把"Token"看成是代表字符串"NOP"，那么序列将按照定义的那样编码为

```
Command ::= SEQUENCE {
    Parameter INTEGER
}
```

如果一个元素被忽略掉，那么解码器的工作就是向前调整一个元素并以默认值来替代它。显然，默认值必须是已知的，否则解码器不知道用什么值来代替。

3. 选择

选择修改器允许一个元素在给定的实例中可以有多个可能值。实质上来说，解码器将尝试所有期望的解码算法，直到有一个类型符合为止。当一个复杂的容器中包含其他容器时，选择修改器就十分有用了。例如：

```
UserKey ::= SEQUENCE {
   Name        IA5STRING,
   StartDate   UTCTIME,
   Expire      UTCTIME,
   KeyData     CHOICE {
      ECCKey ECCKeyType,
      RSAKey RSAKeyType
   }
}
```

上例大概是一个简单的即允许ECC也允许RSA密钥的公钥证书。这种数据类型的编码等同于任意选择其中一种类型，也就是如下的两个容器的其中之一。

```
ECCUserKey ::= SEQUENCE {
   Name        IA5STRING,
   StartDate   UTCTIME,
   Expire      UTCTIME,
   ECCKey      ECCKeyType,
}
```

```
RSAUserKey ::= SEQUENCE {
   Name        IA5STRING,
   StartDate   UTCTIME,
   Expire      UTCTIME,
   RSAKey      RSAKeyType
}
```

解码器必须接受原始序列“UserKey”并且能够判断出在编码时选择了哪种类型，即使这些选择使用了它们自身的复杂容器。

2.3 ASN.1数据类型

既然我们已经掌握了基本的ASN.1语法，我们就该研究让ASN.1如此有用的ASN.1数据类型和它们的编码。ASN.1针对广泛的应用定义了多种数据类型，但是大部分对于密码学应用没有什么意义，对此我们将不进行讨论。如果读者想掌握ASN.1提供的所有数据类型，建议阅读X.680和X.690系列规范说明。

任何ASN.1编码都以两个字节开始（或者八位位组，含有8个二进制位），不管什么类型，它们都是通用的。第一个字节是类型标识符，它也包含了一些修正位，我们稍后将做讨论。第二个字节是长度。长度在解码时有点复杂，但在实际应用中很容易实现。

我们将要研究以下数据类型。

- 布尔型（Boolean）；
- 八位位组串（OCTET String）；
- 位串（BIT String）；
- IA5String；
- 可打印字符串（PrintableString）；
- 整数（INTEGER）；
- 对象标识符（OBJECT Identifier, OID）；
- 世界协调时（UTCTIME）；
- 空（NULL）；
- 序列、单一序列；
- 集合；
- 单一集合。

这些类型足够实现PKCS#1和ANSI X9.62标准了，但是并没有给开发者带来过多的负担。

2.3.1 ASN.1头字节

头字节（header byte）常位于ASN.1编码的开始，由3部分组成：类别（the classification）、结构化位（the constructed bit）和原始类型（the primitive type）。头字节可以拆分，如图2-2所示。

在ASN.1术语中，从1到8对位进行编号，按照从最低位到最高位的顺序。设置位8就等于C语言中用0x80和该字节进行或（OR）操作；同样地，原始类型值15从位5到位1分别编码为{0,1,1,1,1}。

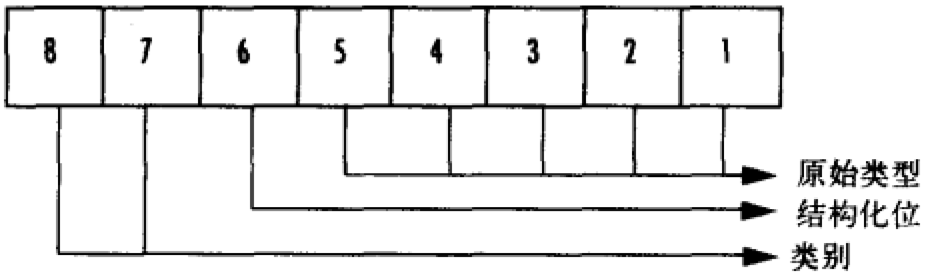


图2-2 ASN.1头字节

1. 类别位

类别位（classification bits）由两个进制位表示，它不修改编码，但描述数据将要解释的上下文。表2-1列出了类别位的设置。

所有的类型当中，通用类别最为常见。类别的选择仅仅是信息的表面或者是一部分。一个有效的DER解码器应当能够分析ASN.1类型而不管它是什么类别。这取决于所使用的协议，它使用解码器来决定根据类别如何处理分析的数据。

表2-1 ASN.1类别

位8	位7	类
0	0	通用（Universal）
0	1	应用（Application）
1	0	上下文特定（Context Specific）
1	1	专用（Private）

2. 结构化位

结构化位（constructed bit）表示一个给定的编码是否是相同类型的多种编码的结构化。当一个应用程序在逻辑上是一个元素，但并不是一次就包含所有组件的元素进行编码时，结构化位就十分有用。结构化元素也是容器类型必需的，因为在逻辑上，它们只是其他元素的集合。

结构化元素有自己的头字节和长度字节，之后是元素各个要素组件的单独编码。也就是说，从它们自身来看，这些要素组件是独立地可解码ASN.1数据类型。

严格来讲，相应于非典型编码规则，容器类是惟一允许使用的结构化数据类型。这是因为对于其他的数据类型，给定的内容，只允许一种编码。我们假设除容器外，其他所有数据类型的结构化位都为0。

3. 原始类型

ASN.1头字节的低5位定义了32种ASN.1原始类型（primitive type）（如表2-2所示）。

表2-2 ASN.1原始类型

代码	ASN.1类型	作用
1	布尔型	存储布尔值
2	整数	存储大整数
3	位串	存储位数组
4	八位位组串	存储字节数组
5	空	预留位（例如在选择修改器中）
6	对象标识符	标识算法及协议
16	序列和单一序列	未分类元素的容器
17	集合和单一集合	已分类元素的容器
19	可打印字符串	ASCII编码（忽略了一些不可打印字符）

（续）

代码	ASN.1类型	作　用
22	IA5String	ASCII编码
23	世界协调时	以统一格式表示的时间

起初乍一看，很奇怪的是规范中没有选择（CHOICE）原始类型。但在前面提到，选择是一个修改器而不是类型；正因为如此，代替它的是对被选中的元素进行编码。每种类型将在本章的后面进行深入的介绍；现在，我们只要知道存在这些类型就可以。

2.3.2 ASN.1长度编码

根据编码的实际长度，ASN.1定义了两种长度编码（length encoding）方法。根据编码的长度和环境，可以用定长或者非定长方式进行编码，并且还可以进一步分割成短编码或者长编码。

在这种情况下，我们只考虑定长编码，而且也要考虑所有类型的短编码和长编码。在基本编码规则中，编码器可以自由地选择短编码或者长编码，但前提是能够完全表示元素的长度。非典型编码规则要求必须选择可以完全表示元素长度的最短编码。编码后的长度并不包括ASN.1头字节和长度字节，以减少负载。

编码的第一个字节代表是短编码还是长编码（如图2-3所示）。

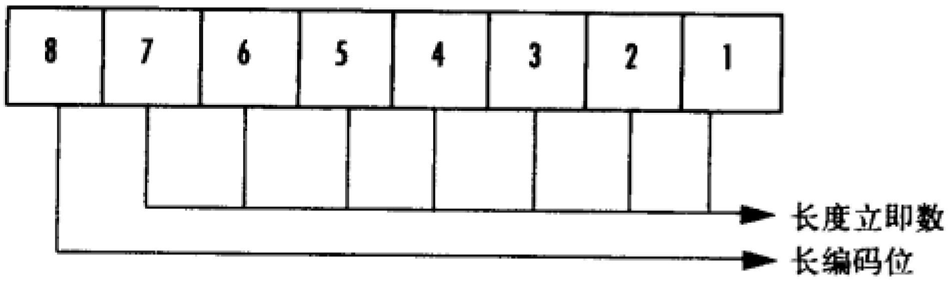


图2-3　长度编码字节

最高位决定编码是短的还是长的，而低7位则形成一个长度立即数。

1. 短编码

在短编码中，负载的长度必须小于128个字节。长度立即数域用来表示负载的长度，这也正是编码大小限制的来源之处。它是对所有长度小于128个字节的强制性编码。例如，对长度65（0x41）进行编码，只需要简单地使用字节0x41即可。由于这个值没有设置最高位，所以解码器可以判断出这是短编码而且长度是65个字节。

2. 长编码

在长编码中，定义了附加的抽象数据来对长度进行编码，它仅适用于所有长度为128个字节或以上的负载。在这种模式下，长度立即数域存储的是为了表示负载长度所需的字节数。进一步讲，它指定了为了编码负载的长度需要多少字节。长度必须以big-endian格式进行编码。

我们通过一个例子来阐述它。为了对长度47 310（0xB8CE）进行编码，我们首先认识到它比127大，所以必须使用长编码格式进行编码。实际长度需要两个八位位组来表示这个值，所以我们必须使用两个长度立即数字节。如图2-3所示，第8位表示长编码模式，并且需要两个字

节来存储长度。因此，第一个字节是0x80 | 0x02或者是0x82。然后使用big-endian格式来存储这些值。所以，全部编码就是0x82 B8 CE。

正如你所看到的，这是非常有效的，因为仅用一个字节就可以表示高达127个字节的长度，这允许对长度高达2^{1016}位的对象进行编码。这是一个相当大的存储范围，直至下个世纪也不会超过这个范围。

根据DER规则，负载长度值的长度必须最小。因此，所有的1字节（即长度立即数是127的长编码）是无效的。一般来说，对于长编码，可以放心地假设长度立即数大于4个字节是无效的，因为很少有密码学协议在一个数据包中会交换超过4GB的数据。

提示 一般来说，禁止超过4字节长度的ASN.1长编码是一种不错的办法，这意味着负载大小会超过4GB。虽然这是一个很好的初始安全检查，但这还不够。建议读者针对输出缓冲区大小，检查完全的解码负载长度，以防止缓冲区溢出攻击。

通常，这种检查应该在负载解码开始实施之前。这避免了在错误编码上花费时间，而且也很容易进行代码审计和复查。

2.3.3 ASN.1布尔类型

布尔（boolean）数据类型用来在编码器或者解码器中，不用很大的开销就可以对简单的布尔值进行编码或者解码。

布尔编码的负载或者是全0或者是全1的单八位位组。头字节以0x01开始，长度字节常为短编码的0x01，内容取决于布尔值的不同取0x00或者0xFF（如表2-3所示）。

表2-3 布尔值编码

布尔值	编码
False	0x01 01 00
True	0x01 01 FF

根据BER规则，真值的编码可以是任何非零值，但是DER要求真值编码为字节0xFF。

2.3.4 ASN.1整数类型

整数（integer）类型表示一个有符号的任意精度的标量，它的编码是可移植的，平台无关的。对正整数的编码相当简单。

存储的实际数值（不同于我们将要看到的负数）分成字节大小的数字，并且以big-endian格式进行存储。例如，对变量$x = 256^k \times x_k + 256^{k-1} \times x_{k-1} + \ldots + 256^0 \times x_0$进行编码，八位位组$\{x_k, x_{k-1}, \ldots, x_0\}$将以递减的顺序从$x_k$到$x_0$进行存储。编码过程规定对于正整数，第一个字节的最高位必须为0。

因此，假设第一个字节大于127（如49 468（0xC13C且0xC1>0x7F），看上去其编码应该是0x02 02 C1 3C，但它的最高位为1，所以应该看成负数。最简单的办法（并且也是正确的）是用前端零字节进行填充。即49 468将编码为0x02 03 00 C1 3C，它是有效的，因为$256^2 \times 0x00 + 256^1 \times 0xC1 + 256^0 \times 0x3C$等于49 468。

负数的编码并不简单。这个过程需要找到一个最小的256的幂，使它比要编码的数的绝对值还大。例如，对－1555进行编码，比1555大的256的最小的幂为$256^2 = 65536$。然后，把这两

个数相加以得到2的补码表示形式，本例中为63 981。实际编码的整数是这个和。

因此，在这个例子中，－1555的ASN.1编码为0x02 02 F9 ED。然后对整数的编码使用两个附加准则，以减小输出的大小。

第一个八位位组的所有位和第二个八位位组的位8必须为：

- 不全为1；
- 不全为0。

整数解码相当容易。如果第一个最高位为0，被编码的值是正的且负载是绝对标量值。如果最高位是1，被编码的值是负的并且要从编码的值中减去下一个最大的256的幂（如表2-4所示）。

尤其要注意128和－128编码的不同。它们都赋值为0x80，但正值需要0x00前缀来区分。

表2-4 INTEGER编码示例

值	编　码
0	0x02 01 00
1	0x02 01 01
2	0x02 01 02
127	0x02 01 7F
128	0x02 02 00 80
−1	0x02 01 FF
−128	0x02 01 80
−32768	0x02 02 80 00
1234567890	0x02 04 49 96 02 D2

2.3.5 ASN.1位串类型

位串（BIT STRING）类型用来以可移植形式表示位数组。除了ASN.1头部之外，还有一个附加的头部用来表示填充数据。

这些位如下进行编码，将第一位放到第一个负载字节的最高位。下一位存储到第一个负载字节的第7位，依此类推。例如，对位串{1, 0, 0, 0, 1, 1, 1, 0}进行编码，需要分别将位8、位4、位3和位2设为1。即{1, 0, 0, 0, 1, 1, 1, 0}编码为0x8E。

编码的第一个字节指定形成一个完整的字节所需要填充的位数。例如位串{1,0,0,1}将转变成{1,0,0,1,0,0,0,0}，其填充数量为4。如果位串为空，填充数为0。有效的填充长度范围为0～7。

负载的长度包括填充数字节和已编码的位。表2-5阐述了上一个位串的编码。

表2-5 BIT STRING编码示例

Code	ASN.1 Type	Use Of
1	Boolean Type	Store Booleans
2	INTEGER	Store Large integers
3	BIT STRING	Store an array of bits
4	OCTET STRING	Store an array of bytes
5	NULL	Place holder(e.g.in a CHOICE)
6	OBJECT IDENTIFIER	Identify algorithms or protocols
16	SEQUENCE and SEQUENCE OF	Container of unsorted elements
17	SET and SET OF	Container of sorted elements
19	PrintableString	ASCII Encoding(omitting several non-printable chars)
22	IA5STRING	ASCII Encoding
23	UTCTIME	Time in a universal format

在图2-4中，我们看到位串{1,0,0,1}的编码为0x03 02 04 90。注意到负载长度是0x02而不是0x01，因为我们把填充字节作为负载的一部分。

解码器通过计算8×负载长度－填充数来得到存储输出所需要的位数。

2.3.6 ASN.1八位位组串类型

除了八位位组串（OCTET STRING）是保存字节（八位位组）数组之外，它和位串类型很相似。这种类型的编码相当简单。像任何其他类型一样对ASN.1头部进行编码，然后直接将八位位组复制过去就可以了。

例如对八位位组串{FE, ED, 6A, B4}进行编码，首先存储类型0x04，接着是长度0x04，然后是字节本身0xFE ED 6A B4。这不能再简单了。

2.3.7 ASN.1空类型

空（NULL）类型实际上是“占位符”，它是含有空白选项的选择修改器所特有的。例如，考虑如下的序列（SEQUENCE）：

```
MyAccount ::= SEQUENCE {
   Name       IA5String,
   Group      IA5String,
   Credentials CHOICE {
               rsaKey       RSAPublicKey,
               passwdHash   OCTET STRING,
               none         NULL
   },
   LastLogin UTCTIME,
   …
}
```

在这个结构中，账号的证书应该包含一个RSA密钥或者一个密码散列值或者什么都没有。如果不存在这个空类型，编码器必须选择两者之一，并且将其他类型指定为“空”类型。

空类型的编码是0x05 00。它在DER编码规则中没有负载，然而，从技术上说，在BER编码中必须忽略它的负载。

2.3.8 ASN.1对象标识符类型

对象标识符（OBJECT IDENTIFIER，OID）类型用层次的形式来表示标准规范。标识符树通过一个点分的十进制符号来定义，这个符号以组织、子部分然后是标准的类型和各自的子标识符开始。

例如，MD5散列算法的OID是1.2.840.113549.2.5，这看起来即长又复杂，但实际上这个OID树可以分为“iso（1）member-body（2）US（840）rsadsi（113549）digestAlgorithm（2）md5（5）”。当看到这个OID时，解码程序（并不是解码器本身）能够认识到这是MD5散列算法。

因为这个原因，OID在公钥算法标准中很流行，它指出证书绑定了哪种散列算法。但OID不仅仅局限于散列算法。同样也有公钥算法、分组算法和操作模式的OID。它们是一种高效且可移植的表示数据包中所选算法的形式，并不需要用户（或者第三方用户）来指出算法类型的“魔幻解码”。

这种点分十进制格式是很容易理解的，但下面两种规则除外。

- 第一部分的范围必须为0<=x<=3；
- 如果第一部分小于2，那么第二部分必须小于40。

除此之外，其余部分可以是任何正的无符号数。它们的大小通常小于32位，但并不一定都是这样。

对各个部分的编码有点复杂，但仍然是可以容易理解的。前两部分如果定义为x.y，那么它们将合成一个字40x+y，其余部分单独作为一个字进行编码。

每个字首先被分割为最少数量的没有头零数字的7位数字。这些数字以big-endian格式进行组织，并且一个接一个地组合成字节。除了编码的最后一个字节外，其他所有每个字节的最高位（位8）都为1。例如，30 331分割成7位的数字之后为{1, 108, 123}，设置最高位之后变成{129, 236, 123}。如果该字只有一个7位数字，那么最高位为0。

把它应用于MD5 OID，首先将点分十进制形式转化成字数组。这样，1.2.840.113549.2.5就为{42, 840, 113549, 2, 5}，然后进一步将其分割为带有最高位的7位数字，即{{0x2A}, {0x86, 0x48}, {0x86, 0xF7, 0x0D}, {0x02}, {0x05}}。因此，MD5的全编码为0x06 08 2A 86 48 86 F7 0D 02 05。

解码更简单，除了要把第一个字分割成两部分，首先用第一个字模40，余数就是第二部分。把它从字中减去，然后再除以40，就可以得到第一部分。

2.3.9 ASN.1序列和集合类型

序列（SEQUENCE）和单一序列（SEQUENCE OF）以及相应的集合（SET）和单一集合（SET OF）类型叫做“结构”类型或者叫简单容器。它们是一种用来把相关数据元素收集为一个独立的可解码元素的简单方法。

- 按照X.690规范，序列具有如下性质。
- 编码是结构化的；

编码的内容应由ASN.1序列类型定义列表中的所有数据类型值的完全编码所组成，并且按照它们出现的顺序进行编码，除非这些类型被可选（OPTIONAL）或者默认（DEFAULT）关键字修改器所引用。

结构化的意思是说位6必须设置，这使得序列头字节的值变成由0x10到0x30。结构化编码是一种简单的嵌套编码。例如，考虑如下的序列。

```
User ::= SEQUENCE {
   ID      INTEGER,
   Active BOOLEAN
}
```

当对值{32, TRUE}进行编码时，我们首先给字节0x30以表明这是一个结构化的序列。接着，我们给出负载的长度，即整数和布尔类型的编码长度，共6个字节即0x06。然后开始结构化部分。我们将整型编码为0x02 01 20，布尔型编码为0x01 01 FF。因此整个编码为0x30 06 02 01 20 01 01 FF。在ASN.1文档中，它们使用空白来表明编码的属性。

```
0x30 06
        02 01 20
        01 01 FF
```

如果使用下面的嵌套结构，这种符号表示特别有用。

```
Account ::= SEQUENCE {
   User SEQUENCE {
```

```
        Name     PrintableString,
        Group    PrintableString,
        Credential SEQUENCE {
           PasswdHash    OCTET STRING,
           RSAKey        RSAPublicKey  OPTIONAL
        }
    },
    LastOn    UTCTIME,
    Valid     BOOLEAN
}
```

当给定序列{{ "tom", "users", {0x01 02 03 04 05 06 07 08}}, "060416180000Z", TRUE}，它将被编码为如下形式：

```
Account       0x30 2C
User                   30 18
Name                         13 03 74 6F 6D
Group                        13 05 75 73 65 74 75
Credential                   30 0A
PasswdHash                         04 08 01 02 03 04 05 06 07 08
LastOn                 17 0D 30 36 30 34 31 36 31 38 30 30 30 30 30 5A
Valid                  01 01 FF
```

在这个例子中，我们可以清楚地看到编码中的序列嵌套。特别地，也可以看到我们省略了作为用户证书一部分的RSA密钥可选项。注意，负载长度是组成序列的所有部分的长度。

提示 和OpenSSL库一同安装的openssl命令行程序提供了一种把用DER编码的文件转化成可阅读的输出的简单办法。这对于调试你自己的ASN.1程序很有用，你可以和这个已知是正确的第三方工具进行比较。下面的命令读一个文件并且显示可解码的元素。

```
openssl asn1parse -inform der -in $INFILE -i
```

其中$INFILE是要读的文件。如果你想从管道中读入，可以省略"-in $INFILE"。

```
tom@bigbox ~ $ openssl asn1parse -inform DER -in test.der -i
    0:d=0  hl=3 l= 159 cons: SEQUENCE
    3:d=1  hl=2 l=  13 cons:  SEQUENCE
    5:d=2  hl=2 l=   9 prim:   OBJECT            :rsaEncryption
   16:d=2  hl=2 l=   0 prim:   NULL
   18:d=1  hl=3 l= 141 prim:  BIT STRING
```

第一列指出了在文件中的偏移，"d"表示嵌套深度，"hl"指定头部的长度，"l"为负载长度。"cons"和"prim"指出它是结构化的（容器）还是原始类型（头字节的位6），最后一个字指出其原始类型。

从缩进中我们可以看出序列从偏移3开始，它是第一个序列中的一个元素。类似地，对象（OBJECT）和空（NULL）元素是第二个序列的元素。这里我们也可以看到OpenSSL把对象重组为"rsaEncryption"，在本例中它是一个公钥。

1. 单一序列

单一序列类似于序列，除非此序列只是一种类型的容器。这是ASN.1中数组的等价定义。单一序列的编码包含0或者以编码器指定的顺序（前面提到过）对ASN.1类型列表中的类型编

码。单一序列使用同样的头字节0x30，用来表示它是序列家族的一员。这意味着解码器即可以读取序列也可以读取单一序列类型。

2. 集合

集合是一种类似于序列的结构化类型，但它的头字节是0x31而不是0x30，且其成员编码的顺序并不是集合定义的顺序。严格来讲，对BER规则来说，发送者是可以判断出顺序的。意思是说，如果没有将集合的顺序事先发送给接收者，那么，集合不能包含两个相同的类型。在DER编码规则中，是按照类型值的顺序进行升序排列。如果两个元素有相同的类型，则由已提交集合的原始顺序决定"加时赛规则"。即这个重复类型的第一次出现为"胜者"。

考虑前面的序列，其集合编码为：

```
User ::= SET {
   ID     INTEGER,
   Active BOOLEAN

}
```

当对值{32，TRUE}进行编码时，我们首先给出字节0x31来表示这是一个结构化的集合。从前面可知其长度为6个字节，在集合中这是不变的。因此，我们现在给出字节0x06。现在，根据DER规则，首先按照它们的类型进行排序。因为布尔的类型为0x01，整数型的类型为0x02，所以布尔类型是第一个。因此，完整的编码为0x31 06 01 01 FF 02 01 20。

下面所列的集合包含一个冲突。

```
User ::= SET {
   ID       INTEGER,
   Active   BOOLEAN,
   LogCount INTEGER
}
```

在这个例子中，ID和LogCount有相同的类型：整数型。实例{32, TRUE, 1023}的编码以头字节0x31开始，接着是长度字节，本例中为0x0A。接下来，对布尔型进行编码，因为它的类型在数值上小于0x01 01 FF。ID和LogCount为相同的类型，但是ID首先出现，所以接下来存储它的编码，为0x02 01 20。最后，LogCount存储为0x02 02 3F FF。因此，完整的编码为0x31 0A 01 01 FF 02 01 20 02 02 3F FF。

3. 单一集合

单一集合是类似于单一序列的集合。根据BER规则，元素的顺序无关紧要。在DER规则中，与按照类型排序不同的是，我们按照组成元素的DER编码升序排列。考虑整数数组{1，10007，0，20，−300}，它们各自的编码如表2-6所示。

当把表2-6中的编码升序排列时，我们发现表2-7中所示的列表为它们编码后的顺序。

表2-6 整数数组

数值	编码
1	0x02 01 01
10007	0x02 02 27 17
0	0x02 00
20	0x02 01 14
−300	0x02 02 FE D4

表2-7 排序后的整数集合

数值	编码
0	0x02 00
1	0x02 01 01
20	0x02 01 14
10007	0x02 02 27 17
−300	0x02 02 FE D4

对于排序后的数据，首先要注意的是对一个给定的单一集合，其编码首先以它们的长度排序，然后再以它们的负载。这是由于ASN.1所使用的长度编码的原因。其次，注意到对于给定的我们要排序的数据，排序并不是都有逻辑意义的。−300出现在后面，甚至在0的后面，即使作为一个整数，它表示更小的一个数值。

对于DER编码进行排序的目的是为了简单地确保编码是可决定的（或者是按照ASN.1规范进行区分），而不管编码器输入的顺序如何。

因此，这个单一集合数组的完整编码为0x31 10 02 00 02 01 01 02 01 14 02 02 27 17 02 02 FE D4。

2.3.10 ASN.1可打印字符串和IA5String类型

可打印字符串（PrintableString）和IA5String类型定义了一种独立于本地代码页和字符集定义，在任何平台上都可以将ASCII字符串编码为可读字符串的可移植方法。

可打印字符串编码对象是ASCII集合的一个有限子集，这个子集包括32（空格）、39（单引号）、40～41、43～58、61、63以及65～122。这些范围之外的数值都是无效的并且应该报错。可打印字符串的意思是在不改变显示文字流的情况下，能够在大多数的终端下打印出来的字符。因为这个原因，它忽略了值小于32的ASCII码。

IA5String的编码对象是ASCII集合中的大多数值，它包括NUL、BEL、TAB、NL、LF、CR以及在32和126之间的ASCII码值，包括32和126。通常，在没有过滤的情况下使用TTY来显示IA5String是不安全的，因为它允许那些已编码的数据做一些诸如清屏、替换字符之类的事情，这些事情取决于所使用的终端类型。

除了那些限制和不同的头字节之外，这两种类型的编码和八位位组串相似。可打印字符串的头字节是0x13，IA5String为0x16。例如“Hello World”，它将编码为0x13 0B 48 65 6D 6D 6F 20 57 6F 72 6D 64。

注意 *检查这些编码值是否在相应数据类型的允许范围内，即是编码器也是解码器的责任。*

2.3.11 ASN.1世界协调时类型

世界协调时（UTCTIME）定义了一种相对于GMT时间的标准时间（以日期）编码。ASN.1的前期草案允许时间差值（时区）和折叠编码（例如省略秒）。对于DER来说，有6种可能的方法对秒为0的日期进行编码。从X.690的2002草案开始，所有的UTC编码都应该使用“YYMMDDHHMMSSZ”这种格式，分别表示年、月（1～12）、日（0～31）、时（0～23）、分（0～59）及秒（0～59）。

“Z”遗留自初始UTCTIME，如果没有“Z”，那么，就允许两种附加组“[+/−]hh‘mm’”，其中“hh”和“mm”分别为与GMT的时差和分差（正值或者负值）。如果有“Z”，则时间是以Zulu或者GMT时间表示的。

字符串的编码按照IA5String编码规则进行转换（使用ASCII字符集），其头字节为0x17而不是0x16。例如July 4,2003 at 11:33 and 28 seconds的编码为“030704113328Z”，这个字符串的编

码为0x17 0D 30 33 30 37 30 34 31 31 33 33 32 38 5A。

2.4 实现

现在我们将要考虑如何实现ASN.1编码器和解码器。幸运的是大部分的ASN.1类型都是原始类型，而且很容易处理。如果想要一个用户友好的API，那么开发结构化类型会有些困难。我们将尽最大的努力来编写ASN.1程序，以便最终代码可以最大限度地重复使用。

所有的ASN.1程序都可以在源代码库中的“ch2”目录中找到。在目录里有一系列的C源代码文件，一个单独的包含了协议类型的H头文件，一个GNU Makefile，它可以使用GCC将这些文件构建成一个包。

我们研究的第一个程序实现获得、读取和编码ASN.1编码的长度。所有其他的ASN.1类型也有这种逻辑概念，正因为如此，才可以在可能的时候重用这些代码。

2.4.1 ASN.1长度程序

第一个程序简单地返回编码的长度，包括头、长度字节和负载。

```
der_length.c:
#include "asn1.h"
unsigned long der_length(unsigned long payload)
{
   unsigned long x;

   if (payload > 127) {
      x = payload;
      while (x) {
        x >>= 8;
        ++payload;
      }
   }

   return payload + 2;
}
```

这个函数接受以负载的长度作为输入，返回最终编码的大小。这个函数适合于所有的类型，且其负载长度是预知的，即原始（非结构化）类型。注意，对于位串类型，调用函数须将负载长度加上填充计数器字节以保证它可以正确地计算。

这个函数对编码器很有用，因为它允许调用者知道最终输出的长度，而且当缓冲区溢出发生时会产生错误报告。倘若合理地使用这个检查，那么在编码开始之前进行这个简单的检查可以省去后面很多的麻烦。

对ASN.1头部进行编码并不复杂，但我们需要对编码稍微变形一下。因为这个函数将在编码器中使用，所以它可以在编码过程中允许编码器调用一些上下文数据。

```
der_put_header_length.c:
void der_put_header_length(unsigned char **out,
                           unsigned primitive_type,
                           unsigned long payload_length,
                           unsigned long *outlen)
{
```

```
006     unsigned char *ptr;
007     unsigned long x, y, pl;
008
009     ptr = *out;
010
011     /* store header */
012     *ptr++ = primitive_type;
013
014     /* encode payload length */
015     if (payload_length < 128) {
016        *ptr++ = payload_length;
017     } else {
018        /* determine length of length */
019        x = payload_length;
020        y = 0;
021        while (x) {
022          ++y;
023          x >>= 8;
024        }
025
026        /* store length of length  */
027        *ptr++ = 0x80 | y;
028
029        /* align length on 32-bit boundary, we assume y < 5 */
030        x  = y;
031        pl = payload_length;
032        while (x < 4) {
033           pl <<= 8;
034           ++x;
035        }
036
037        /* store it */
038        while (y--) {
039           *ptr++ = (pl >> 24) & 0xFF;
040        }
041     }
042
043    /* get stored size */
044    *outlen = (ptr - *out) + payload_length;
045
```

这个函数可以将ASN.1的头部及其长度存储在一个调用函数指定的缓冲区中。指向缓冲区的指针实际上是将一个指针传递给另一个指针。这允许函数可以修改输出指针并且在头部编码之后还可以继续编码。

这个函数假设负载小于4GB，这是很实际的。当长度大于127个字节时，对长度的编码是通过将非零最高字节移出负载长度来完成的（第30行代码）。它可以使得用big-endian格式对长度进行提取，按照ASN.1的要求，先提取最高字节，然后把长度向左移动8位。

注意，这个函数并不检查输出缓冲区长度，调用函数需要在调用这个函数之前确保输出缓冲区足够大。这个函数维护了一个输出指针的内部拷贝*ptr*，并且在函数退出之前将其复制给输出。这样做不是为了效率的原因，而仅仅是为了让代码更容易阅读。

现在我们可以对头部进行编码，同样也要能对其进行解码。在理论上，解码函数应该有一个相似的协议类型，因为它几乎是编码的逆方向。在这个函数中，我们先介绍我们的第一个函

数，它有一种失败的情形。

```
der_get_header_length.c:
unsigned long der_get_header_length(unsigned char **in,
                                    unsigned long   inlen,
                                    unsigned        *primitive_type,
                                    unsigned long   *payload_length)
{
   unsigned long x, y;
   unsigned char *ptr;

   ptr = *in;

   /* ensure inlen is at least two */
   if (inlen < 2) {
      return -1;
   }

   /* get the type and first length byte */
   *primitive_type = *ptr++;
   y = *ptr++;

   if (y < 128) {
      *payload_length = y;
   } else {
      y &= 0x7F;  /* strip off bit 8 */

      /* simple safety check */
      if (y > 4 || (2 + y) > inlen) {
         return -1;
      }

      /* read in the length */
      x = 0;
      while (y--) {
         x = (x << 8) | *ptr++;
      }
      *payload_length = x;
   }

   *in = ptr;
   return 0;
}
   }

  /* get stored size */
  *outlen = (ptr - *out) + payload_length;

   /* update the output pointer */
  *out = ptr;

}
```

这个函数对ASN.1头部进行解码，如果解码成功，它将把原始类型和负载长度存储在调用函数传入的指针中。这个函数引入了我们的错误报告机制，当有错误发生时会返回非零值。

这个函数也需要调用函数传入输入的长度来做为一个参数。这用来防止缓冲区越界，攻击

者可以用它来读取栈中（或者是堆中，这取决于是从哪里读数据的）的数据。除了针对读错误而进行的输入长度检查之外，这个函数也做了完整的检查，并要求所有的负载长度都小于4GB。

我们也注意到这个函数会存储最终输出长度。这很有用，因为这确保任何使用它的编码器都能对调用者（编码器的）自动设置输出长度。

注意 对于位串类型，返回的负载长度包括填充计数器字节。

2.4.2 ASN.1原始编码器

既然我们已经可以分析ASN.1头部了，那么我们就可以开始处理ASN.1类型。每个原始类型都提供3个程序：一个长度判定函数、一个编码函数和一个解码函数。

长度判定函数叫做der_XYZ_length()，它的输入是将要进行编码的数据（在需要输入的时候），并且返回完整编码的长度。这些函数在判定编码是否可以在提供的缓冲区中进行是十分有用的。在处理结构化类型时也很有用。

编码和解码函数不言而喻是相关的，分别是der_XYZ_encode()和der_XYZ_decode()。编码和解码函数都有可能失败，所以调用者应该检查它的返回值。如果输入是无效的，或者输出缓冲区太小，编码函数会失败。解码函数也具有同样的失败原因。

1. 布尔编码

我们按照ASN.1类型值的顺序开始第一个类型。幸运地是，这意味着第一个类型是布尔类型，而它又是最容易处理的。

长度判定函数相当简单。

```
der_boolean_length.c:
unsigned long der_boolean_length(void)
{
   return 3;
}
```

正如我们所看到的，这个函数只有4行。不管是真值还是假值，所有的布尔类型编码都是3个字节的长度。值得注意的是，这个长度判定函数有可能会失败。在这种情况下，我们使用0长度作为错误指示器。对布尔类型而言，所有的输入都是有效的。

```
der_boolean_encode.c:
#include "asn1.h"
int der_boolean_encode(int            bool,
                       unsigned char *out,
                       unsigned long *outlen)
{
   /* check output size */
   if (der_boolean_length() > *outlen) {
      return -1;
   }

   /* store header and length  */
   der_put_header_length(&out, ASN1_DER_BOOLEAN, 1);

   /* store payload */
   *out = (bool == 0) ? 0x00 : 0xFF;

```

```
017     /* finished ok */
018     return 0;
019  }
```

这个函数以ASN.1 DER格式对布尔类型进行编码。它把*bool*作为要编码的布尔值。可以用0来表示假，用任何非零值表示真。

这个函数使用der_boolean_length()函数（第7行代码）来判断输出缓冲区是否足够大，以存放编码数据。即使知道编码始终是3个字节，我们仍然使用这个函数，因为它是所有其他ASN.1编码器都将使用的模式。这是一个值得养成的习惯，尤其是考虑到这个函数并不是一个性能瓶颈。

在检查完输出缓冲区的长度之后，我们使用der_put_header_length()函数（第12行代码）来存储ASN.1头部。这一行也用到了ASN1_DER_BOOLEAN符号，它是在“asn1.h”头文件中定义的。这个调用函数为我们存储了ASN.1头部和长度字节。

当从函数der_put_header_length()返回后，输出指针已经修改为指向ASN.1头部的后一个字节。这样，在不知道ASN.1头部长度的情况下，也可以进行负载编码。取决于输入是假或真，负载是简单的0x00或者0xFF。

现在，为了解码ASN.1 DER布尔值，我们以一个相似的“stock”函数继续。

```
der_boolean_decode.c:
001  #include "asn1.h"
002  int der_boolean_decode(unsigned char *in,
003                         unsigned long  inlen,
004                         int            *bool)
005  {
006     unsigned type;
007     unsigned long payload_length;
008     int      ret;
009
010     /* decode header */
011     ret = der_get_header_length(&in, inlen,
012                                 &type, &payload_length);
013     if (ret < 0) {
014        return ret;
015     }
016
017     /* payload must be 1 byte and 0x00 or 0xFF */
018     if (type           != ASN1_DER_BOOLEAN ||
019         payload_length != 1 ||
020         (in[0] != 0x00 && in[0] != 0xFF)) {
021         return -2;
022     }
023
024     /* decode payload */
025     *bool = (in[0] == 0xFF) ? 1 : 0;
026
027     return 0;
028  }
```

我们试图做的第一件事是进行头部解码并保证能得到一个负载长度。函数调用der_get_header_length()（第11行代码）在一个函数调用中为我们做了这件事。如果函数失败，则返回一个小于0的值，所以我们可能直接给调用函数返回错误代码。像在编码函数中一样，输出指

针将修改为指向负载的第一个字节。

在分析完ASN.1头部之后，我们将验证类型、长度和负载是否都有效。如果这些内容是无效的，将返回−2表示出现了解码错误。一旦验证完数据包，将把布尔值存回调用函数传入的指针里。

2. 整数编码

当数值为正时，整数的编码相当容易。在正数的情况下，编码只是简单的以big-endian格式存储的整数字节编码。对于负数，我们需要找到一个大于该负数绝对值的256的某个幂，然后对两者之和进行编码。

ASN.1规范并没有对要编码的整数大小的下限进行限制，因此，整数的范围和能够编码的长度一样（小于128个字节的长度）。但是，此时我们还没有处理大整数的方法，因此，我们将人工限制我们自己使用C语言中*long*数据类型。建议读者注意此项功能的局限性，以便合理地支持整数类型。

在进行编码和解码之前，需要开发一些处理整数的辅助函数。这些函数在典型的大数库中很常见而且很容易理解。

```
int_help.c:
#include "asn1.h"

int count_bits(long num)
{
   int x;
   x = 0;
   while (num) { ++x; num >>= 1; }
   return x;
}
```

这个函数返回整数的位数。它假设输入的整数是正的（或者至少无符号扩展），且简单地将整数右移直到0为止。

```
int count_lsbs(long num)
{
   int x;
   x = 0;
   if (!num) return 0;
   while (!(num&1)) { ++x; num >>= 1; }
   return x;
}
```

这个函数计算整数的连续0最低位数目。它也假设输入的整数没有符号扩展。注意，这个函数是one-based，而不是zero-based。例如，如果这个数为10_2，返回值为1，100_2返回2，等等。

```
void store_unsigned(unsigned char *dst, long num)
{
   int x, y;
   unsigned char t;

   x = y = 0;
   while (num) {
       dst[x++] = num & 255;
```

```
028            num >>= 8;
029        }
030
031        /* reverse */
032        --x;
033        while (y < x) {
034           t = dst[x]; dst[x] = dst[y]; dst[y] = t;
035           --x; ++y;
036        }
037    }
```

这个函数以big-endian字节格式存储一个正整数。第一个循环（第26行代码）从该数中提取字节。此时，*dst*数组存储该整数的little-endian格式。这是因为我们在每一次循环迭代时存储的是它的最低字节。

第二个循环（第33行代码）把这些字节进行交换，使得它们按照big-endian格式存储。这通过从两端开始，交换字节和内部移动来完成。

```
039    long read_unsigned(unsigned char *dst, unsigned long len)
040    {
041       long tmp;
042
043       tmp = 0;
044       while (len--) {
045          tmp = (tmp << 8) | *dst++;
046       }
047       return tmp;
048    }
049
```

这个函数读取字节并将它们存储为一个整数。即使它是向左移动的，但它总是以big-endian格式来解释输入。在这个函数中不需要交换。

这4个函数是我们在这一步需要提供的所有的函数，它们用来在编码器和解码器中合理地处理ASN.1整数类型。我们可以开始长度函数了。现在我们引入一个新的函数“paylen”，它只判断编码的负载长度，而不是整个编码的长度。

paylen函数在判断最终输出的大小上可以派上用场，并且告诉头编码器使用多少负载长度。

```
#include "asn1.h"
unsigned long der_integer_paylen(long num)
{
   unsigned long x, y, pad, paylen;

   if (num >= 0) {
      /* it's positive */
      /* count # of bits  */
      x = count_bits(num);

      /* if the 8th bit is set we pad */
      if ((x & 7) == 0) {
         pad = 1;
      } else {
         pad = 0;
      }
```

```
017
018         /* round count up to the next byte */
019         x = x + ((8 - x) & 7);
020         paylen = (x >> 3) + pad;
021      } else {
022         /* it's negative */
023         x = count_bits(-num);
024         y = count_lsbs(-num);
025
026         /* round count up to the next byte */
027         paylen = x + ((7 - x) & 7);
028         paylen = (paylen >> 3) + 1;
029
030         /* if lsbs+1==bits and bits mod 8 == 0 reduce by 1 */
031         if ((y+1)==x && !(x & 7)) {
032            --paylen;
033         }
034      }
035      return paylen;
036   }
```

这个函数计算整数的负载长度。首先，我们判断该数是否为正数（第6行代码），并以不同的方法分别处理正数和负数。在是正数的情况下，我们简单地计算出表示的位数，如果需要则进行填充（第12行代码），然后转向下一个字节（第19行代码）。当第一个字节的最高位为1时，则需要进行填充。这时，我们必须增加一个头0字节以确保编码将被解释为正数。

对于负数，我们既需要位数也需要头0最低位。首先，我们转到下一个字节并加上一个附加字节。之所以需要这个附加字节，是因为我们要计算与256的幂的和。

这对于$-(256^k)/2$形式的数字会产生异常，它会产生冗余的0xFF 80字节编码，而这在ASN.1 DER规范中是禁止的。

```
039   unsigned long der_integer_length(long num)
040   {
041      /* get payload length */
042      return der_length(der_integer_paylen(num));
043   }
```

现在我们简单地通过使用针对负载长度的der_length()函数来返回整个DER编码长度。

```
der_integer_encode.c:
#include "asn1.h"
int der_integer_encode(          long  num,
                        unsigned char *out,
                        unsigned long *outlen)
{
   /* check output size */
   if (der_integer_length(num) > *outlen) {
      return -1;
   }

   /* encode header */
   der_put_header_length(&out, ASN1_DER_INTEGER,
                         der_integer_paylen(num), outlen);

```

PDG

```
015      /* store number */
016      if (num >= 0) {
017         /* leading msb? */
018         if ((count_bits(num) & 7) == 0) {
019            *out++ = 0x00;
020         }
021         store_unsigned(out, num);
022      } else {
023         /* find power of 256 greater than it */
024         int x, y, z;
025         long tmp;
026
027         x = count_bits(-num);
028         y = count_lsbs(-num);
029         z = ((x + ((7 - x) & 7)) >> 3) + 1;
030
031         /* handle special case */
032         if ((y+1)==x && !(x & 7)) {
033             --z;
034         }
035
036         /* get our constant */
037         tmp = 1L << (z << 3);
038
039         /* encoding */
040         store_unsigned(out, tmp + num);
041      }
042
043      return 0;
044   }
```

像布尔编码函数一样，我们首先检查是否有足够的空间来使用der_integer_length（）函数，接着对ASN.1头部进行编码。此时，我们根据要编码的整数是正还是负将下面的程序分成两个路径。

当整数为正时，我们只要输出所需的所有头0字节（取决于第一个字节的MSB），然后以字节格式存储整数（第21行代码）。

当整数为负时，我们需要找到比整数的绝对值大的最小256的幂。这和der_integer_length（）函数中的实现大致相同。为了精确地计算数的幂，我们使用简单的左移运算（第37行代码）。最后，我们保存它们的和，并以2的补码的格式进行编码。

```
der_integer_decode.c:
#include "asn1.h"

int der_integer_decode(unsigned char *in,
                       unsigned long  inlen,
                       long *num)
{
   unsigned type;
   unsigned long payload_length;
   long     tmp;
   int      ret;

   /* decode header */
```

```
013      ret = der_get_header_length(&in, inlen,
014                                  &type, &payload_length);
015      if (ret < 0) {
016         return ret;
017      }
018
019      if (type != ASN1_DER_INTEGER) {
020         return -2;
021      }
022
023      /* read in the value */
024      tmp = read_unsigned(in, payload_length);
025
026      /* if the leading byte has 0x80 set it's negative */
027      if (in[0] & 0x80) {
028         /* it's negative */
029         *num = tmp - (1L << (payload_length << 3));
030      } else {
031         *num = tmp;
032      }
033
034      return 0;
035   }
```

这个函数是对整数进行解码的。首先对头部进行解码，读入该整数，然后根据负载第一个字节的最高位来分析它。如果该位为1（第27行代码），该数为负数，那么我们要对该数加上256的某个幂。

3. 位串编码

位串是对每个字节按照由最高位到最低位进行编码的位数组，它含有8个已编码位。它可以有多至7个的填充位，以确保在长度上是8的某个整数倍。前面已提到，ASN.1头部里的负载长度包括用来表示填充长度的单个字节。

```
der_bitstring_length.c:
#include "asn1.h"
unsigned long der_bitstring_length(unsigned long nbits)
{
   unsigned long bytes;

   /* get # of payload bytes */
   bytes = 1 + (nbits >> 3) + ((nbits & 7) ? 1 : 0);

   return der_length(bytes);
}
```

这个函数通过统计位数和加上填充计数器字节来计算位串的长度。

```
der_bitstring_encode.c:
#include "asn1.h"
int der_bitstring_encode(unsigned char *bits,
                         unsigned long  nbits,
                         unsigned char *out,
                         unsigned long *outlen)
{
   unsigned long bytes, bitbuf, bitcnt, tcnt;
```

```
008
009      /* check output size */
010      if (der_bitstring_length(nbits) > *outlen) {
011         return -1;
012      }
013
014      /* store header and length  */
015      bytes = 1 + (nbits >> 3) + ((nbits & 7) ? 1 : 0);
016      der_put_header_length(&out, ASN1_DER_BITSTRING,
017                            bytes, outlen);
018
019      /* store padding count */
020      *out++ = (8 - nbits) & 7;
021
022      /* accumulate bits */
023      tcnt    = nbits;
024      bitbuf = bitcnt = 0;
025      while (tcnt--) {
026         bitbuf = (bitbuf << 1) | (*bits++ & 1);
027         if (++bitcnt == 8) {
028            *out++ = bitbuf;
029            bitbuf = bitcnt = 0;
030         }
031      }
032
033      /* pad any remaining bytes */
034      if (nbits & 7) {
035         bitbuf <<= ((8 - nbits) & 7);
036         *out++ = bitbuf;
037      }
038
039      return 0;
040   }
```

在常规的初始代码之后，编码过程继续读取所有提供的位，并按照每次8个位将其压缩成一个字节（第26行代码）。当一个字节全满时（第27行代码），就将它存储到输出缓冲区，并且重置该字节缓冲区（第28～29行代码）。

所有的剩余位将模仿填充位的插入方法进行左移（第35行代码），并将其存储到输出中。

der_bitstring_decode.c:

```
#include "asn1.h"
int der_bitstring_decode(unsigned char *in,
                         unsigned long  inlen,
                         unsigned char *out,
                         unsigned long *outlen)
{
   unsigned type;
   unsigned long payload_length, nbits, bitbuf, bitcnt;
   int      ret;

   /* decode header */
   ret = der_get_header_length(&in, inlen,
                               &type, &payload_length);
   if (ret < 0) {
      return ret;
```

```
016        }
017
018        if (type != ASN1_DER_BITSTRING || payload_length < 1) {
019           return -2;
020        }
021
022        /* get # of bits */
023        nbits = ((payload_length - 1) << 3) - in[0];
024        ++in;
025
026        /* too many? */
027        if (nbits > *outlen) {
028           return -1;
029        }
030        *outlen = nbits;
031
032        /* start decoding */
033        bitbuf = *in++;
034        bitcnt = 8;
035
036        while (nbits--) {
037           *out++    = (bitbuf & 0x80) >> 7;
038           bitbuf <<= 1;
039           if (--bitcnt == 0) {
040               bitbuf = *in++;
041               bitcnt = 8;
042           }
043        }
044        return 0;
045     }
```

像往常一样，我们对ASN.1头部进行解码以得到其类型和长度。接下来，我们验证类型是否正确以及负载至少要有一个字节（第18行代码）。由于编码时有填充计数器字节，所以负载至少要有一个字节。

接下来，我们比较输出长度并将位数赋值给调用函数所提供的输出长度变量（第27～30行代码）。我们通过把负载乘以8（左移3位）并减去填充位数来计算输出的位数。

类似于编码函数，我们设置一个位缓冲区来处理输入。bitbuf变量保存当前正在处理的字节，bitcnt保存剩下的位数。这些位依次从最高位读入，然后存入输出数组中（第36～42行代码）。

4. 八位位组串编码

八位位组串的编码是目前为止最容易处理的。解码函数没有无效输入，负载长度等于输入的长度。

der_octetstring_length.c:

```
#include "asn1.h"
unsigned long der_octetstring_length(unsigned long noctets)
{
   return der_length(noctets);
}
```

这个相当易懂的函数计算一个给定八位位组数的八位位组串的DER编码长度。

der_octetstring_encode.c:

```
001  #include "asn1.h"
002  int der_octetstring_encode(unsigned char *octets,
003                             unsigned long  noctets,
004                             unsigned char *out,
005                             unsigned long *outlen)
006  {
007     /* check output size */
008     if (der_octetstring_length(noctets) > *outlen) {
009        return -1;
010     }
011
012     /* store header and length  */
013     der_put_header_length(&out, ASN1_DER_OCTETSTRING,
014                           noctets, outlen);
015
016     /* store bytes */
017     memcpy(out, octets, noctets);
018
019     return 0;
020  }
```

到目前为止，这是最简单的非平凡编码函数。它检查输出长度（第8行代码），取出头部（第13行代码），然后简单地把输入复制到输出（第17行代码）。

提示 大部分看上去很简洁的C函数，如memcpy、memcmp、malloc及free等嵌入到开发平台中是很危险的。在许多跨编译器的开发环境中，可能都没有一个完整的C函数库或者任何一个库。

堆是另一个棘手的问题。在许多平台上，释放内存空间是有严格规定的，这常意味着它将由应用程序来管理。

LibTom项目采用了一种简单的办法来解决这种问题，即对通用函数使用C预处理宏。例如XMALLOC的定义，它可以用下面的代码来定义其默认为malloc。

```
#ifndef XMALLOC
#define XMALLOC malloc
#endif
```

在本例中，如果在这段被送往预处理程序处理之前XMALLOC就已经定义过了，那么，它可能会将XMALLOC重定向为“调用”其他函数。例如：

```
CFLAGS="-DXMALLOC=mymalloc" gcc myprog.c -lmylib -o myprog
```

现在，在程序里你可以简单的调用XMALLOC而不是直接使用malloc ()。例如：

```
unsigned char *buffer = XMALLOC(buffer_size);
```

这种技术可以用于应用程序中的其他标准C函数。

为了完整起见，下面是八位位组串的解码函数。

der_octetstring_decode.c:

```
#include "asn1.h"
int der_octetstring_decode(unsigned char *in,
                           unsigned long  inlen,
                           unsigned char *out,
```

```
                                    unsigned long *outlen)
{
   unsigned type;
   unsigned long payload_length;
   int      ret;

   /* decode header */
   ret = der_get_header_length(&in, inlen,
                                    &type, &payload_length);
   if (ret < 0) {
      return ret;
   }

   if (type != ASN1_DER_OCTETSTRING) {
      return -2;
   }

   /* check output size */
   if (payload_length > *outlen) {
      return -1;
   }

   /* copy out */
   *outlen = payload_length;
   memcpy(out, in, payload_length);

   return 0;
}
```

5. 空值编码

空类型的编码是所有编码中最简单的。它只能用一种方法来表示并且是定长的。为了完整起见，如下是空值的程序。

der_null_length.c:

```
unsigned long der_null_length(void)
{
  return 2;
}
```

der_null_encode.c:

```
#include "asn1.h"
int der_null_encode(unsigned char *out,
                         unsigned long *outlen)
{
   if (*outlen < 2) return -1;

   out[0]  = ASN1_DER_NULL;
   out[1]  = 0x00;
   *outlen = 2;

   return 0;
}
```

der_null_decode.c:

```
#include "asn1.h"
```

```
int der_null_decode(unsigned char *in,
                    unsigned long inlen)
{
   if (inlen != 2 || in[0] != ASN1_DER_NULL || in[1] != 0x00) {
      return -2;
   }
   return 0;
}
```

在编码和解码函数中，我们都简单地直接读取和存储数据而没有使用辅助函数。通常来讲，这是一种不好的编程实践，同样也不是一种好习惯。但是在本例中，不这么做会让函数有更多行的代码，所以我们就例外一次。读者要注意到即使我们知道空的类型值是0x05，我们仍然要使用符号ASN1_DER_NULL。这当然是一种值得推荐的编程实践。

6. **对象标识符编码**

对象标识符（OID）类型的编码有些像正整数的编码，不同的是它使用7位数而不是8位数。OID是一系列正的无符号数（称为字）的集合，因为它们是相互紧接的，所以它们的编码也是相关联的。每个字节的最高位代表它是否是OID中一个给定的字的最后7位数。

一个OID的前两个字指定了OID参照的是哪种标准实体。它们比较特殊，第一个字值的范围在0～3之间，第二个字必须在0～39之间。通过把第一个字乘以40再加上第二个字，它们被编码为一个单独的无符号数。其余的字各自独立进行编码。

der_oid_length.c:

```
#include "asn1.h"

unsigned long der_oid_paylen(unsigned long *in,
                             unsigned long inlen)
{
   unsigned long wordbuf, y, z;

   /* form first word */
   wordbuf = in[0] * 40 + in[1];
   z       = 0;
   for (y = 1; y < inlen; y++) {
      if (wordbuf == 0) {
         ++z;
      } else {
         /* count the # of 7 bit digits in wordbuf */
         while (wordbuf) {
            ++z;
            wordbuf >>= 7;
         }
      }
      if (y < inlen - 1)
         wordbuf = in[y + 1];
   }
   return z;
}
```

上面是负载长度函数的实现，这是因为找出负载长度是很重要的，所以最好用一个单独的函数来实现。编码的第一个字实际上是前面提到的两个输入的级联。为了保持逻辑上的一致，我们使用一个缓冲区*wordbuf*来保存当前字，这个字的编码决定了它所包含的7位数的个数。

值为0的字是一种例外（第12行代码），因为它们没有非零7位数，但是它们仍然需要至少一个字节的输出来表示。在这个例外处理之后，我们开始从变量*wordbuf*中提取7位数。如果这不是算法的最后一次循环（第21行代码），我们取下一个字然后进行迭代。

安全警告！ 在der_oid_paylen()函数的第21行代码中，在第22行取下一个字之前，我们检查了是否已处于循环的结束。在大部分平台上，即使我们处于最后一次迭代，第22行代码也不会造成任何问题。这可以通过设定一个比正在读取的数组大的栈或者堆来避免问题的产生。

但是，从稳定性和安全性的角度来看，它仍然有会产生问题的场合。在某些x86模块中，会限制一个段只有很小的大小，或者读取数组结尾后面的数据会超过页界（产生页错误）。因此，一种简单的读操作完全有可能使一个程序崩溃。

当执行读操作时，也有可能会产生安全问题。有可能wordbuf是位于栈中的，这意味着不管数组的结尾之后是什么（也许是一个秘密密钥），它也是位于栈中的。虽然这个单独的函数不能直接泄露秘密，但其他的没有改写那部分栈的函数可能轻易的泄露栈中的内容。

这种安全概念也存在于资源管理中，对于开发者来说，意识到这个问题是相当重要的。

```
027   unsigned long der_oid_length(unsigned long *in,
028                                unsigned long inlen)
029   {
030      if (in[0] > 3 || in[1] > 39) {
031         return 0;
032      }
033      return der_length(der_oid_paylen(in, inlen));
034   }
```

这个函数高度依赖于负载长度函数。它首先检查前两个字以确定它们都在允许的范围内，然后计算其DER编码长度。

```
der_oid_encode.c:
#include "asn1.h"
int der_oid_encode(unsigned long *in,
                   unsigned long  inlen,
                   unsigned char *out,
                   unsigned long *outlen)
{
   unsigned long x, y, z, t, wordbuf, mask;
   unsigned char tmp;
   /* check output size */
   x = der_oid_length(in, inlen);
   if (x == 0 || x > *outlen) {
      return -1;
   }

   /* store header and length  */
   der_put_header_length(&out, ASN1_DER_OID,
                         der_oid_paylen(in, inlen),
                         outlen);

```

至此，这是标准的编码代码。因为OID输入可能是无效的，所以我们检查了是否有溢出以及输入是否有效（第11行代码），这可以通过der_oid_length()是否返回0来看出。这里也要注意der_oid_paylen()的用法。重用它的功能代码可以省去大量的麻烦，因为我们没有必要重复造轮子。

```
020      /* encode words */
021      wordbuf = in[0] * 40 + in[1];
022      for (y = 1; y < inlen; y++) {
```

这里，编码的第一个字是前两个字的组合。也要注意循环是从1开始而不是0。

```
023          if (wordbuf) {
024             /* mark current spot and clear mask */
025             x    = 0;
026             mask = 0x00;
027             while (wordbuf) {
028                out[x++]  = (wordbuf & 0x7F) | mask;
029                wordbuf >>= 7;
030                mask     |= 0x80;
031             }
```

此时，如果*wordbuf*初始时不为0，数组out[0...*x*-1]将包含这个字的little-endian格式编码。变量*mask*的使用可以使我们能够对所有存储的7位数设置最高位，当然不包括第一个7位数。

```
033             /* now swap x-1 bytes */
034             z = 0;
035             t = x--;
036             while (z < x) {
037                 tmp = out[z]; out[z] = out[x]; out[x] = tmp;
038                 ++z; --x;
039             }
```

像整型编码处理程序一样，我们把输出转化为big-endian格式。变量*t*保存该字的长度。由于是从0到*x*-1进行交换的，而不是从0到*x*，所以我们将其减1。

```
041             /* move pointer */
042             out += t;
043          } else {
044             *out++ = 0x00;
045          }
046          if (y < inlen - 1) {
047             wordbuf = in[y+1];
048          }
049      }
050      return 0;
051   }
```

我们也通过存储一个字节0x00来处理值为0的情形。因为最高位不为1，所以解码函数会将这个字解释为一个单一的字节。在这个字的编码完成以后，如果循环还没有结束的话就取下一个字，否则返回。

```
der_oid_decode.c:
#include "asn1.h"
int der_oid_decode(unsigned char *in,
                   unsigned long  inlen,
```

```
                         unsigned long *out,
                         unsigned long *outlen)
{
   unsigned type;
   unsigned long payload_length, wordbuf, y;
   int       ret;

   /* decode header */
   ret = der_get_header_length(&in, inlen,
                               &type, &payload_length);
   if (ret < 0) {
      return ret;
   }

   /* check type and enforce a minimum output size of 2 */
   if (type != ASN1_DER_OID || *outlen < 2) {
      return -2;
   }
```

至此，这是相当标准的解码逻辑代码。我们同样对输出的初始大小进行检查。因为所有的OID都至少含有前两个字，所以输出长度不能小于2。

注意，在解码之前我们没有计算字数。这是因为我们没有对每个将要解码的字分配资源。如果不得不在解码之前分配资源（堆），那也是很简单的。这可以防止泄露，因为资源的使用通常是有潜在危险的。

```
   wordbuf = 0;
   y       = 0;
   while (payload_length--) {
      wordbuf = (wordbuf << 7) | (*in & 0x7F);
```

像整型的解码函数一样，我们读入字节并将其左移。其中，我们只使用低7位。这个while循环包含了所有的负载字节，且理论上说，当while循环结束时，所有的OID字都已经处理过。

```
      if (!(*in++ & 0x80)) {
         /* last 7 bit digit */
         if (y == 0) {
            /* first words */
            out[0] = wordbuf / 40;
            out[1] = wordbuf % 40;
            y      = 2;
         } else {
            if (y < *outlen) {
               out[y++] = wordbuf;
            } else {
               return -2;
            }
         }
         wordbuf = 0;
      }
   }

   /* store size */
   *outlen = y;
```

```
047
048        return 0;
049    }
```

如果发生没有设置最高位的情况，那么这个字节就是给定字的最后一个字节。在这个实现中，变量y是当前已存储的字数。如果它为0，那么就知道将要解码的即是输出的前两个个字。我们提取这两个字（第31行和第32行代码）并且更新y的值。否则就检查输出的长度，如果可能的话就进行存储操作。

虽然ASN.1规范没有这么说，但是我们将字限制为unsigned long数据类型。这意味着它们不能在大小上超过2^{32}-1。在实际应用中，这使得代码更简单而且不会和任何密码学标准冲突。

7. 可打印的和IA5字符串编码

可打印的IA5字符串的编码非常类似于八位位组串的编码，不同的是它的输入有范围限制，而且在存储之前必须通过一种方便移植的机制来转化。这是因为不同的平台处理代码页的方式不同。例如，在大多数的ASCII平台上（如大部分的英文版Linux和Windows），

```
char c = 'a';
```

可能等于

```
char c = 97;
```

但在其他平台上并非都如此。为便于这种处理，我们需要一个可以被C编译器以本地代码页解释的表，然后将其转化为整型以进行编码和解码。

为简单起见，我们只演示可打印字符串的编码，IA5String的编码可以类推（IA5处理程序可以从书的网站上获得）。

```
der_printablestring_length.c:
#include "asn1.h"

static const struct {
   int code, value;
} printable_table[] = {
{ ' ', 32 }, { '\'', 39 }, { '(', 40 }, { ')', 41 },
{ '+', 43 }, { ',', 44 }, { '-', 45 }, { '.', 46 },
{ '/', 47 }, { '0', 48 }, { '1', 49 }, { '2', 50 },
{ '3', 51 }, { '4', 52 }, { '5', 53 }, { '6', 54 },
{ '7', 55 }, { '8', 56 }, { '9', 57 }, { ':', 58 },
{ '=', 61 }, { '?', 63 }, { 'A', 65 }, { 'B', 66 },
{ 'C', 67 }, { 'D', 68 }, { 'E', 69 }, { 'F', 70 },
{ 'G', 71 }, { 'H', 72 }, { 'I', 73 }, { 'J', 74 },
{ 'K', 75 }, { 'L', 76 }, { 'M', 77 }, { 'N', 78 },
{ 'O', 79 }, { 'P', 80 }, { 'Q', 81 }, { 'R', 82 },
{ 'S', 83 }, { 'T', 84 }, { 'U', 85 }, { 'V', 86 },
{ 'W', 87 }, { 'X', 88 }, { 'Y', 89 }, { 'Z', 90 },
{ 'a', 97 }, { 'b', 98 }, { 'c', 99 }, { 'd', 100 },
{ 'e', 101 }, { 'f', 102 }, { 'g', 103 }, { 'h', 104 },
{ 'i', 105 }, { 'j', 106 }, { 'k', 107 }, { 'l', 108 },
{ 'm', 109 }, { 'n', 110 }, { 'o', 111 }, { 'p', 112 },
{ 'q', 113 }, { 'r', 114 }, { 's', 115 }, { 't', 116 },
{ 'u', 117 }, { 'v', 118 }, { 'w', 119 }, { 'x', 120 },
{ 'y', 121 }, { 'z', 122 }, };
```

使用这张映射表可以将字符由本地页转化为ASCII数字表示形式。这张表用static和const两个关键字来定义，以避免该表搅乱符号命名空间，并使得它的定义不必出现在代码中。

提示 const和static这两个关键字在嵌入式开发中对符号名的管理和内存的使用非常有用。

const关键字告诉编码器（在大部分情况下）把数据放到应用程序的代码段中。默认是在代码段保存一份，然后在运行时再复制一份到内存中（如GNU工具生成的bss段）。这意味着这张表即占用RAM也占用ROM。

static关键字告诉编码器不要将其设为全局变量。这对于那些相似算法中可能会引起冲突的通用函数定义很有用。这也有助于将污染减少到最小，尤其是在使用一些第三方库的时候相当重要。

```
int der_printable_char_encode(int c)
{
   int x;
   for (x = 0; x <
        (int)(sizeof(printable_table) /
              sizeof(printable_table[0])); x++) {
       if (printable_table[x].code == c) {
          return printable_table[x].value;
       }
   }
   return -1;
}

int der_printable_value_decode(int v)
{
   int x;
   for (x = 0; x <
        (int)(sizeof(printable_table) /
              sizeof(printable_table[0])); x++) {
       if (printable_table[x].value == v) {
          return printable_table[x].code;
       }
   }
   return -1;
}
```

这两个函数分别是将字符转化为数值和将数值转化为字符。如果找到了相应的字符或者值，它们将返回其编码（或者解码），并且在失败时返回－1。

```
unsigned long der_printablestring_length(unsigned char *in,
                                         unsigned long inlen)
{
   unsigned long x;

   for (x = 0; x < inlen; x++) {
     if (der_printable_char_encode(in[x]) == -1) {
        return 0;
     }
   }

   return der_length(inlen);
```

```
}
```

这个函数非常像der_octetstring_length()的副本，不同的是它必须首先保证输入都是有效的。

der_printablestring_encode.c:

```
001   #include "asn1.h"
002   int der_printablestring_encode(unsigned char *in,
003                                  unsigned long  inlen,
004                                  unsigned char *out,
005                                  unsigned long *outlen)
006   {
007      unsigned long x;
008
009      /* check output size */
010      x = der_printablestring_length(in, inlen);
011      if (x == 0 || x > *outlen) {
012         return -1;
013      }
014
015      /* store header and length  */
016      der_put_header_length(&out, ASN1_DER_PRINTABLESTRING,
017                            inlen, outlen);
018
019      for (x = 0; x < inlen; x++) {
020         *out++ = der_printable_char_encode(*in++);
021      }
022
023      return 0;
024   }
```

在编码前我们先检查输入的长度和有效性（第10行代码）。用长度函数来检查有效性是很重要的，因为后面的代码都假设输入是有效的。编码是通过简单地转化函数调用来完成的（第20行代码）。

der_printablestring_decode.c:

```
#include "asn1.h"
int der_printablestring_decode(unsigned char *in,
                               unsigned long  inlen,
                                 unsigned char *out,
                               unsigned long *outlen)
{
   unsigned type;
   unsigned long payload_length, x;
   int      ret;

   /* decode header */
   ret = der_get_header_length(&in, inlen,
                               &type, &payload_length);
   if (ret < 0) {
      return ret;
   }

   if (type != ASN1_DER_PRINTABLESTRING) {
      return -2;
   }

```

```
022        if (payload_length > *outlen) {
023           return -1;
024        }
025        *outlen = payload_length;
026
027        for (x = 0; x < payload_length; x++) {
028            ret = der_printable_value_decode(*in++);
029            if (ret == -1) {
030                return -2;
031            }
032            *out++ = ret;
033        }
034        return 0;
035    }
```

不同于编码函数，解码函数在处理之前并不知道输入是否有效。在对每个字节进行解码时（第28行代码），它同时检查返回值。如果返回值为−1，则表示输入的字节无效并且必须返回一个错误代码。

既然我们已经看过了可打印字符串的编码，那么IA5的编码基本上也是一样的。仅有的不同是我们使用不同的表。

```
der_ia5string_length.c:
static const struct {
   int code, value;
} ia5_table[] = {
{ '\0', 0 },{ '\a', 7 }, { '\b', 8 }, { '\t', 9 },
{ '\n', 10 }, { '\f', 12 }, { '\r', 13 }, { ' ', 32 },
{ '!', 33 }, { '"', 34 }, { '#', 35 }, { '$', 36 },
{ '%', 37 }, { '&', 38 }, { '\'', 39 }, { '(', 40 },
{ ')', 41 }, { '*', 42 }, { '+', 43 }, { ',', 44 },
{ '-', 45 }, { '.', 46 }, { '/', 47 }, { '0', 48 },
{ '1', 49 }, { '2', 50 }, { '3', 51 }, { '4', 52 },
{ '5', 53 }, { '6', 54 }, { '7', 55 }, { '8', 56 },
{ '9', 57 }, { ':', 58 }, { ';', 59 }, { '<', 60 },
{ '=', 61 }, { '>', 62 }, { '?', 63 }, { '@', 64 },
{ 'A', 65 }, { 'B', 66 }, { 'C', 67 }, { 'D', 68 },
{ 'E', 69 }, { 'F', 70 }, { 'G', 71 }, { 'H', 72 },
{ 'I', 73 }, { 'J', 74 }, { 'K', 75 }, { 'L', 76 },
{ 'M', 77 }, { 'N', 78 }, { 'O', 79 }, { 'P', 80 },
{ 'Q', 81 }, { 'R', 82 }, { 'S', 83 }, { 'T', 84 },
{ 'U', 85 }, { 'V', 86 }, { 'W', 87 }, { 'X', 88 },
{ 'Y', 89 }, { 'Z', 90 }, { '[', 91 }, { '\\', 92 },
{ ']', 93 }, { '^', 94 }, { '_', 95 }, { '`', 96 },
{ 'a', 97 }, { 'b', 98 }, { 'c', 99 }, { 'd', 100 },
{ 'e', 101 }, { 'f', 102 }, { 'g', 103 }, { 'h', 104 },
{ 'i', 105 }, { 'j', 106 }, { 'k', 107 }, { 'l', 108 },
{ 'm', 109 }, { 'n', 110 }, { 'o', 111 }, { 'p', 112 },
{ 'q', 113 }, { 'r', 114 }, { 's', 115 }, { 't', 116 },
{ 'u', 117 }, { 'v', 118 }, { 'w', 119 }, { 'x', 120 },
{ 'y', 121 }, { 'z', 122 }, { '{', 123 }, { '|', 124 },
{ '}', 125 }, { '~', 126 } };
```

对于感兴趣的读者，完全可能写出一个节约存储空间和代码空间的程序。如果给表加上第三个参数，表示该符号属于哪种编码类型（即IA5或者可打印的字符串），那么只要一个编码/

解码函数就可以了，这个函数可以使用一个附加参数来说明将要使用哪种目标代码。

为了简单易懂，演示的程序代码是各自独立实现的。它们相当小，而且从代码风格上来讲，也不是代码大小的决定者。

8. 世界协调时编码

世界协调时编码已经在2002规范中做了简化，现在只包含一种格式。时间按格式“YYMMDDHHMMSSZ”编码为一个字符串，其中每个部分使用两个数字。

年的编码是取最后两个数字。超过2069的年份将被编码成1970（当然，到那时我们可以将软件重写，并作为2070年来处理）。尽管Y2K（千年虫问题）已经爆发，但它们仍使用日期的最后两个数字。

字符串的实际编码是使用ASCII码，幸运的是，我们只要有可打印字符串编码的知识就足够了。

为了有效地处理日期数据，这们需要一些结构来存储参数。我们可以把它们6个作为参数直接传递给函数。但一种更有用的方式是使用结构（在处理序列类型时我们也将看到）。这个结构在ASN.1头文件中定义。

```
asn1.h:
/* UTCTIME */
typedef struct {
   int year,
       month,
       day,
       hour,
       min,
       sec;
} UTCTIME;
```

我们已将其定义为一个类型，因此我们可以简单地传递UTCTIME，而不用传递“struct UTCTIME”。

```
der_utctime_length.c:
#include "asn1.h"
unsigned long der_utctime_length(void)
{
   return 15;
}
```

我非常感谢修改的ASN.1规范！

```
der_utctime_encode.c:
#include "asn1.h"

static int putnum(int val, unsigned char **out)
{
   unsigned char *ptr;
   int            h, l;

   if (val < 0) return -1;

   ptr = *out;
   h   = val / 10;
```

```
    l   = val % 10;

    if (h > 9 || l > 9) {
       return -1;
    }

    *ptr++ = der_printable_char_encode("0123456789"[h]);
    *ptr++ = der_printable_char_encode("0123456789"[l]);
    *out = ptr;

    return 0;
}
```

这个函数把一个整数保存为一个两位数的ASCII码形式表示。这个函数成功执行的条件是该数字必须在0～99范围内。它修改输出指针，我们即将看到在下一个函数中马上就产生了作用。

我们重用der_printable_char_encode()将整数转化为要求的ASCII码。

```
int der_utctime_encode(      UTCTIME *in,
                       unsigned char *out,
                       unsigned long *outlen)
{
    /* check output size */
    if (der_utctime_length() > *outlen) {
       return -1;
    }

    /* store header and length  */
    der_put_header_length(&out, ASN1_DER_UTCTIME, 13, outlen);

    /* store data */
    if (putnum(in->year % 100, &out) ||
        putnum(in->month, &out) ||
        putnum(in->day, &out) ||
        putnum(in->hour, &out) ||
        putnum(in->min, &out) ||
        putnum(in->sec, &out)) {
        return -1;
    }

    *out++ = der_printable_char_encode('Z');

    return 0;
}
```

在头部编码这个标准问题之后，我们将按顺序保存日期和时间域。我们大量地使用C语言中的双或线（||）。尤其是，它总是从左到右来实现，并将在出现第一个非零返回值时取消，而不继续向下进行处理。

意思是说，比如在对月份进行编码时，程序失败了，那么它就不会对剩下的数据进行编码，而直接转到大括号外继续运行。

```
der_utctime_decode.c:
#include "asn1.h"

static int readnum(unsigned char **in, int *dest)
```

```
{
   int x, y, z, num;
   unsigned char *ptr;

   num = 0;
   ptr = *in;
   for (x = 0; x < 2; x++) {
       num *= 10;
       z    = der_printable_value_decode(*ptr++);
       if (z < 0) {
          return -1;
       }
       for (y = 0; y < 10; y++) {
           if ("0123456789"[y] == z) {
               num += y;
               break;
           }
       }
       if (y == 10) {
          return -1;
       }
   }

   *dest = num;
   *in   = ptr;

   return 0;
}
```

这个函数把两个字节解码成其所表示的数值。它再次借用可打印字符串的处理函数将字节转化为字符。它把数值返回到“dest”域并通过返回代码给出错误。我们将在解码函数中使用同样的办法来叠加一系列的读入值，这也是为什么把结果保存在调用函数提供的指针中的重要原因。

```
int der_utctime_decode(unsigned char *in,
                             unsigned long  inlen,
                                   UTCTIME *out)
{
   unsigned type;
   unsigned long payload_length;
   int      ret;

   /* decode header */
   ret = der_get_header_length(&in, inlen,
                                    &type, &payload_length);
   if (ret < 0) {
      return ret;
   }
   if (type != ASN1_DER_UTCTIME || payload_length != 13) {
      return -2;
   }

   if (readnum(&in, &out->year)  ||
       readnum(&in, &out->month) ||
       readnum(&in, &out->day) ||
       readnum(&in, &out->hour) ||
```

```
055          readnum(&in, &out->min) ||
056          readnum(&in, &out->sec)) {
057          return -1;
058       }
059
060       /* must be a Z here */
061       if (der_printable_value_decode(in[0]) != 'Z') {
062          return -2;
063       }
064
065       /* fix up year */
066       if (out->year < 70) {
067          out->year += 2000;
068       } else {
069          out->year += 1900;
070       }
071
072       return 0;
073    }
```

这里我们使用叠加的办法（第51行～56行代码）来分析输入，并一次一个地将解码域存储到输出结构中。在对所有的域解码结束之后，我们检查所需的‘Z’（第61行代码），然后将年域调整为合适的世纪（第66行～第77行代码）。

9. *序列编码*

序列类型连同集合和单一集合类型是目前为止最需要集成开发的类型。它们需要所有的ASN.1函数，当然也包括它们自身的。

为了简洁起见，我们省略了对集合类型编码的介绍，而只介绍简单的序列处理函数。建议读者自己去探索包括处理集合类型的函数的完整列表。

在对序列进行编码之前，我们要做的第一件事是用C语言的某种形式来表示它。在本例中，我们使用一个结构数组来表示序列中的元素。

```
asn1.h:
typedef struct {
   int            type;
   unsigned long  length;
   void          *data;
} asn1_list;
```

这个结构对应于一个序列元素类型（它也用于集合类型中）。*type*表示元素的ASN.1类型，*length*表示值的长度或者大小，*data*是一个指向给定数据类型的本地表示的指针。这种结构类型的数组描述了序列中的所有元素。

根据ASN.1类型的不同，长度和数据域有着不同的含义。表2-8描述了它们的用法。

表2-8 ASN.1_List类型的定义

ASN.1类型	"Data"的含义	"Length"的含义
布尔型	指向一个int的指针	忽略
整数型	指向一个long的指针	忽略
位串	指向一个unsigned char数组	位的个数

（续）

ASN.1类型	"Data"的含义	"Length"的含义
八位位组串	指向一个unsigned char数组	八位位组的个数
空	忽略	忽略
对象标识符	指向一个unsigned long数组	OID中的字数
IA5字符串	指向一个unsigned char数组	字符个数
可打印字符串	指向一个unsigned char数组	字符个数
世界协调时	指向一个UTCTIME结构	忽略
序列	指向一个asn1_list数组	列表中的元素个数

取决于是编码还是解码，长度域起到一个双重角色的作用。除序列类型之外，所有长度域没有被忽略的类型，其长度域都指定了解码时最大的输出大小。它会由解码函数修改为解码对象的实际长度（按照你传递给编码函数的类型）。

例如，如果你指定了一个长度为16的位串元素，并且解码函数将长度域设为7，这意味着解码函数读取了一个7位的位串。

我们也定义了一个便于在运行时环境中创建列表的宏，但首先让我们继续序列函数。

```
der_sequence_length.c:
#include "asn1.h"
unsigned long der_sequence_paylen(asn1_list     *list,
                                  unsigned long  length)
{
   unsigned long i, paylen, x;

   for (i = paylen = 0; i < length; i++) {
      switch (list[i].type) {
```

序列和集合函数的关键部分是一个巨大的针对所有类型的开关语句。这里，我们使用ASN1_DER_*值，它确实对应了所有的ASN.1类型。但是，最后还是使用ASN1_DER_*符号，而不是使用直接的数值。

```
         case ASN1_DER_BOOLEAN:
            paylen += der_boolean_length();
            break;
         case ASN1_DER_INTEGER:
            paylen +=
              der_integer_length(*((long *)list[i].data));
            break;
```

这里我们可以看到使用结构的.data成员来访问将要编码的实际数据。我们使用了C语言中的这样一个事实，void指针可以用任何其他指针类型来替代而不必违反标准。数据指针所指向的实际数据决定了我们用什么类型来编码。在本例中，它是一个long型。

```
         case ASN1_DER_BITSTRING:
            paylen += der_bitstring_length(list[i].length);
            break;
```

这里我们可以看到结构中.length成员的使用。长度的含义是说对于给定类型的单位个数。在这个位串例子中，它的意思是要编码的位数，又比如在OID中，长度意味着OID编码中字的个数。

```
            case ASN1_DER_OCTETSTRING:
                paylen += der_octetstring_length(list[i].length);
                break;
            case ASN1_DER_NULL:
                paylen += der_null_length();
                break;
            case ASN1_DER_OID:
                x = der_oid_length(list[i].data, list[i].length);
                if (x == 0) return 0;
                paylen += x;
                break;
            case ASN1_DER_PRINTABLESTRING:
                x = der_printablestring_length(list[i].data,
                                               list[i].length);
                if (x == 0) return 0;
                paylen += x;
                break;
            case ASN1_DER_IA5STRING:
                x = der_ia5string_length(list[i].data,
                                         list[i].length);
                if (x == 0) return 0;
                paylen += x;
                break;
            case ASN1_DER_UTCTIME:
                paylen += der_utctime_length();
                break;
            case ASN1_DER_SEQUENCE:
                x = der_sequence_length(list[i].data,
                                        list[i].length);
                if (x == 0) return 0;
                paylen += x;
                break;
            default:
                return 0;
        }
    }
    return paylen;
}

unsigned long der_sequence_length(asn1_list     *list,
                                  unsigned long  length)
{
    return der_length(der_sequence_paylen(list, length));
}
```

der_sequence_encode.c:

```
#include "asn1.h"
int der_sequence_encode(asn1_list     *list,
                        unsigned long  length,
                        unsigned char *out,
                        unsigned long *outlen)
{
    unsigned long i, x;
    int           err;

    /* check output size */
```

```
    if (der_sequence_length(list, length) > *outlen) {
       return -1;
    }

    /* store header and length  */
    der_put_header_length(&out, ASN1_DER_SEQUENCE,
                          der_sequence_paylen(list, length),
                          outlen);

    /* now encode each element */
    for (i = 0; i < length; i++) {
        switch (list[i].type) {
           case ASN1_DER_BOOLEAN:
               x = *outlen;
               err = der_boolean_encode(*((int *)list[i].data),
                                         out, &x);
               if (err < 0) return err;
               out += x;
               break;
           case ASN1_DER_INTEGER:
               x = *outlen;
               err = der_integer_encode(*((long *)list[i].data),
                                         out, &x);
               if (err < 0) return err;
               out += x;
               break;
           case ASN1_DER_BITSTRING:
               x = *outlen;
               err = der_bitstring_encode(list[i].data,
                                           list[i].length,
                                           out, &x);
               if (err < 0) return err;
               out += x;
               break;
           case ASN1_DER_OCTETSTRING:
               x = *outlen;
               err = der_octetstring_encode(list[i].data,
                                             list[i].length,
                                             out, &x);
               if (err < 0) return err;
               out += x;
               break;
           case ASN1_DER_NULL:
               x = *outlen;
               err = der_null_encode(out, &x);
               if (err < 0) return err;
               out += x;
               break;
           case ASN1_DER_OID:
               x = *outlen;
               err = der_oid_encode(list[i].data,
                                    list[i].length, out, &x);
               if (err < 0) return err;
               out += x;
               break;
           case ASN1_DER_PRINTABLESTRING:
               x = *outlen;
```

```
                err = der_printablestring_encode(list[i].data,
                                                 list[i].length,
                                                 out, &x);
                if (err < 0) return err;
                out += x;
                break;
            case ASN1_DER_IA5STRING:
                x = *outlen;
                err = der_ia5string_encode(list[i].data,
                                           list[i].length,
                                           out, &x);
                if (err < 0) return err;
                out += x;
                break;
            case ASN1_DER_UTCTIME:
                x = *outlen;
                err = der_utctime_encode(list[i].data, out, &x);
                if (err < 0) return err;
                out += x;
                break;
            case ASN1_DER_SEQUENCE:
                x = *outlen;
                err = der_sequence_encode(list[i].data,
                                          list[i].length,
                                          out, &x);
                if (err < 0) return err;
                out += x;
                break;
            default:
                return -1;
        }
    }
    return 0;
}
```

der_sequence_decode.c:

```
#include "asn1.h"
int der_sequence_decode(unsigned char *in,
                        unsigned long  inlen,
                        asn1_list     *list,
                        unsigned long  length)
{
   unsigned type;
   unsigned long payload_length, i, z;
   int      ret;

   /* decode header */
   ret = der_get_header_length(&in, inlen,
                               &type, &payload_length);
   if (ret < 0) {
      return ret;
   }

   if (type != ASN1_DER_SEQUENCE) {
      return -2;
   }
```

```

    for (i = 0; i < length; i++) {
        switch (list[i].type) {
            case ASN1_DER_BOOLEAN:
                ret = der_boolean_decode(in, payload_length,
                                         ((int *)list[i].data));
                if (ret < 0) return ret;
                z    = der_boolean_length();
                break;
            case ASN1_DER_INTEGER:
                ret = der_integer_decode(in, payload_length,
                                         ((long *)list[i].data));
                if (ret < 0) return ret;
                z    = der_integer_length(*((long *)list[i].data));
                break;
            case ASN1_DER_BITSTRING:
                ret = der_bitstring_decode(in, payload_length,
                                           list[i].data,
                                           &list[i].length);
                if (ret < 0) return ret;
                z    = list[i].length;
                break;
            case ASN1_DER_OCTETSTRING:
                ret = der_octetstring_decode(in, payload_length,
                                             list[i].data,
                                             &list[i].length);
                if (ret < 0) return ret;
                z    = der_octetstring_length(list[i].length);
                break;
            case ASN1_DER_NULL:
                ret = der_null_decode(in, payload_length);
                if (ret < 0) return ret;
                z    = der_null_length();
                break;
            case ASN1_DER_OID:
                ret =
                  der_oid_decode(in, payload_length,
                                 ((unsigned long *)list[i].data),
                                  &list[i].length);
                if (ret < 0) return ret;
                z    =
                  der_oid_length(((unsigned long *)list[i].data),
                                  list[i].length);
                break;
            case ASN1_DER_PRINTABLESTRING:
                ret = der_printablestring_decode(in,
                                                 payload_length,
                                                 list[i].data,
                                                 &list[i].length);
                if (ret < 0) return ret;
                z    = der_printablestring_length(list[i].data,
                                                 list[i].length);
                break;
            case ASN1_DER_IA5STRING:
                ret = der_ia5string_decode(in, payload_length,
                                           list[i].data,
                                           &list[i].length);
```

```
                    if (ret < 0) return ret;
                    z    = der_ia5string_length(list[i].data,
                                                list[i].length);
                    break;
                case ASN1_DER_UTCTIME:
                    ret = der_utctime_decode(in, payload_length,
                                             list[i].data);
                    if (ret < 0) return ret;
                    z    = der_utctime_length();
                    break;
                case ASN1_DER_SEQUENCE:
                    ret = der_sequence_decode(in, payload_length,
                                              list[i].data,
                                              list[i].length);
                    if (ret < 0) return ret;
                    z    = der_sequence_length(list[i].data,
                                              list[i].length);
                    break;
                default:
                    return -1;
            }
            payload_length -= z;
            in             += z;
        }
        return 0;
    }
```

从这些函数中我们可以很容易地指出这个实现中的几个缺陷。首先，这里的程序不支持修改器，例如OPTIONAL、DEFAULT或者CHOICE。其次，解码函数的结构必须完全匹配，否则解码程序会停止。当我们对一个结构进行编码时，调用程序必须理解编码的是什么。编码列表是打算在运行时生成的，而不是在编译时。这可以允许我们有很大层次的灵活性，就像我们怎样使用修改器来解释一样。这个实现的关键方面之一是，允许通过给这3个函数增加最小数量的代码来支持新的ASN.1类型。

即使不管在运行时生成编码序列的能力问题，解码函数也是一个问题。使用这3个函数对猜测型序列进行解码的惟一方法是，使用解码失败的反馈来不断地修改这个列表。例如，一个元素的列表是DEFAULT，然后实际的解码列表才会包含这个元素。如果解码失败，下一个逻辑步骤是尝试不使用默认的现有选项来解码。

对于简单的序列这没有什么问题，但当它们的大小不断增加时（例如一个X.590认证），这种方法完全不合适。一种可行的解决方案是，拥有一个可以理解修改器的完整解码函数并且能够执行所需的前溯（非常类似于任何一种编程语言的分析器，它可以向前寻找权标）。但实际上这比理想的解决方案所要求的更加可行。

10. ASN.1 Flexi解码函数

我们提出这个问题的目的是为了采用一个灵活的解码函数（Flexible Decoder，又叫做The Flexi Decoder），它可以对ASN.1数据进行动态解码，并且可以生成它自己的ASN.1对象链表。只要能识别出已编码的ASN.1类型，我们基本上就可以对任何类型的ASN.1数据进行解码。链表以两个方向进行增长——从左到右，从双亲到孩子。从左到右的链接是针对具有相同深度的

元素，而从双亲到孩子是针对不同的深度（例如序列）。

```
asn1.h
/* Flexi Decoding */
typedef struct Flexi {
   int             type;
   unsigned long   length;
   void           *data;
   struct Flexi   *prev, *next,
                  *child, *parent;
} asn1_flexi;
```

这是我们的Flexi列表，它类似于asn1_list，但不同的是它的内部包含两对链表指针。从调用者的角度来看，它们简单地传入一组字节，然后得到一个对所有已解码元素的链表结构输出。

虽然这种方案能够让我们很容易地对任意的ASN.1数据进行解码，但它仍然不能产生一种马上就能够进行分析的办法。首先，我们给出解码函数，下一节，我们给出一种怎样用它来进行分析的方法。

```
der_flexi_decode.c:
001   #include "asn1.h"
002   #include <stdlib.h>
003   int der_flexi_decode(unsigned char   *in,
004                        unsigned long    inlen,
005                        asn1_flexi     **out)
006   {
007      asn1_flexi     *list, *tlist;
008      unsigned long   len, x, y;
009      int             err;
010
011      list = NULL;
012
013      while (inlen >= 2) {
```

我们以一个空表开始（第11行代码）并稍后对它进行构造。然后我们继续分析数据直至只剩下两个字节为止。这使得我们能够在碰到已识别的ASN.1数据末尾的时候给出错误报告。

```
014         /* make sure list points to a valid node */
015         if (list == NULL) {
016            list = calloc(1, sizeof(asn1_flexi));
017            if (list == NULL) return -3;
018         } else {
019            list->next = calloc(1, sizeof(asn1_flexi));
020            if (list->next == NULL) {
021               der_flexi_free(list);
022               return -3;
023            }
024            list->next->prev = list;
025            list = list->next;
026         }
```

在这个循环中，我们总是至少分配一个链表元素。如果不是第一个元素，我们创建一个邻接节点（第19行代码）并对它进行双向链接（第24行代码），以便我们可以从任何方向遍历链表。

```
028         /* decode the payload length */
```

```
        if (in[1] < 128) {
           /* short form */
           len = in[1];
        } else {
           /* get length of length */
           x   = in[1] & 0x7F;

           /* can we actually read this many bytes? */
           if (inlen < 2 + x) { return -2; }

           /* load it */
           len = 0;
           y   = 2;
           while (x--) {
              len = (len << 8) | in[y++];
           }
        }
```

我们需要知道大部分类型的负载长度。例如，在对一个八位位组串解码之前，我们要知道它的长度以便分配所需的内存。已编码的负载长度就表示这个。对于其他类型如位串和OID，它们最大的大小可以从负载长度中计算得知，这对于达到我们的目的很有好处。

```
        switch (in[0]) {
           case ASN1_DER_BOOLEAN: /* BOOLEAN */
              list->type   = ASN1_DER_BOOLEAN;
              list->length = 1;
              list->data   = calloc(1, sizeof(int));
              if (list->data == NULL) { goto MEM_ERR; }

              err = der_boolean_decode(in, inlen, list->data);
              if (err < 0) { goto DEC_ERR; }

              len = der_boolean_length();
              if (len == 0) { goto LEN_ERR; }
              break;
           case ASN1_DER_INTEGER: /* INTEGER */
              list->type   = ASN1_DER_INTEGER;
              list->length = 1;
              list->data   = calloc(1, sizeof(long));
              if (list->data == NULL) { goto MEM_ERR; }

              err = der_integer_decode(in, inlen, list->data);
              if (err < 0) { goto DEC_ERR; }

              len = der_integer_length(*((long *)list->data));
              if (len == 0) { goto LEN_ERR; }
              break;
           case ASN1_DER_BITSTRING: /* BIT STRING */
              list->type   = ASN1_DER_BITSTRING;
              list->length = (len-1)<<3;
              list->data   =
                  calloc(list->length, sizeof(unsigned char));
              if (list->data == NULL) { goto MEM_ERR; }

              err = der_bitstring_decode(in, inlen,
                                         list->data,
```

```
                                   &list->length);
         if (err < 0) { goto DEC_ERR; }

         len = der_bitstring_length(list->length);
         if (len == 0) { goto LEN_ERR; }
         break;
```

位串的长度通过8*（len－1）来估算，它是所要求的最大的大小。之所以减1是因为负载还包含填充计数器字节。如果我们按8的倍数来分配内存，至多浪费了7个字节。

```
      case ASN1_DER_OCTETSTRING: /* OCTET STRING */
         list->type   = ASN1_DER_OCTETSTRING;
         list->length = len;
         list->data   = calloc(list->length,
                               sizeof(unsigned char));
         if (list->data == NULL) { goto MEM_ERR; }

         err = der_octetstring_decode(in, inlen,
                                      list->data,
                                      &list->length);
         if (err < 0) { goto DEC_ERR; }

         len = der_octetstring_length(list->length);
         if (len == 0) { goto LEN_ERR; }
         break;
      case ASN1_DER_NULL: /* NULL */
         list->type   = ASN1_DER_NULL;
         list->length = 0;
         if (in[1] != 0x00) { goto DEC_ERR; }
         len = 2;
         break;
      case ASN1_DER_OID: /* OID */
         list->type   = ASN1_DER_OID;
         list->length = len;
         list->data   = calloc(list->length,
                               sizeof(unsigned long));
         if (list->data == NULL) { goto MEM_ERR; }

         err = der_oid_decode(in, inlen,
                              list->data,
                              &list->length);
         if (err < 0) { goto DEC_ERR; }

         len = der_oid_length(list->data, list->length);
         if (len == 0) { goto LEN_ERR; }
         break;
```

类似于位串，这里我们不得不估算长度。在本例中，我们利用了这样一个事实，OID的字数不能多于负载的字节数（最小的字编码是1个字节)。这里浪费的内存可能是几个字，对于大多数的环境来讲，这只是很小数量的存储空间。

感兴趣的读者可能会在那里加一个realloc()调用来释放没有使用的字。

```
      case ASN1_DER_PRINTABLESTRING: /* PRINTABLE STRING */
         list->type   = ASN1_DER_PRINTABLESTRING;
         list->length = len;
         list->data   = calloc(list->length,
```

```
                                 sizeof(unsigned char));
            if (list->data == NULL) { goto MEM_ERR; }

            err = der_printablestring_decode(in, inlen,
                                            list->data,
                                            &list->length);
            if (err < 0) { goto DEC_ERR; }

            len = der_printablestring_length(list->data,
                                            list->length);
            if (len == 0) { goto LEN_ERR; }
            break;
         case ASN1_DER_IA5STRING: /* IA5 STRING */
            list->type   = ASN1_DER_IA5STRING;
            list->length = len;
            list->data   = calloc(list->length,
                                 sizeof(unsigned char));
            if (list->data == NULL) { goto MEM_ERR; }

            err = der_ia5string_decode(in, inlen,
                                      list->data,
                                      &list->length);
            if (err < 0) { goto DEC_ERR; }

            len = der_ia5string_length(list->data,
                                      list->length);
            if (len == 0) { goto LEN_ERR; }
            break;
         case ASN1_DER_UTCTIME: /* UTC TIME */
            list->type   = ASN1_DER_UTCTIME;
            list->length = 1;
            list->data   = calloc(1, sizeof(UTCTIME));
            if (list->data == NULL) { goto MEM_ERR; }

            err = der_utctime_decode(in, inlen, list->data);
            if (err < 0) { goto DEC_ERR; }

            len = der_utctime_length();
            if (len == 0) { goto LEN_ERR; }
            break;
         case ASN1_DER_SEQUENCE: /* SEQUENCE */
            list->type   = ASN1_DER_SEQUENCE;
            list->length = len;

            /* we know the length of the objects in
               the sequence, it's len bytes */
            err = der_flexi_decode(in+2, len, &list->child);
            if (err < 0) { goto DEC_ERR; }
            list->child->parent = list;

            /* len is the payload length, we have
               to add the header+length */
            len += 2;
            break;
         default:
            /* invalid type, soft error */
            inlen = 0;
```

```
184                len    = 0;
185                break;
186            }
187            inlen -= len;
188            in    += len;
```

当遇到一个不能识别的类型时（第181行代码），我们并不给出一个严重错误，只是简单地重设长度，结束解码程序然后退出。我们需要将长度设为0（第184行代码）来避免负载长度的减操作产生小于0的结果。

我们使用der_*_length()函数来计算解码类型的长度并且将该值存回*len*。在到达结尾处时（第187行代码），我们知道输入指针该向上移动多少以及该减去多少剩下的输入长度。

```
189        }
190
191        /* we may have allocated one more than we need */
192        if (list->type == 0) {
193            tlist       = list->prev;
194            free(list);
195            list        = tlist;
196            list->next = NULL;
197        }
```

此时我们已经给链表添加了多余的节点。这是由出现一个0类型来判定的，它并不是一个有效的ASN.1类型。如果发生这种情况，我们取前一个链接点，释放叶子节点并修改相应的指针（第193行～196行代码）。

```
198
199        /* rewind */
200        while (list->prev) {
201           list = list->prev;
202        }
203
204        *out = list;
205        return 0;
206     MEM_ERR:
207        der_flexi_free(list);
208        return -3;
209     DEC_ERR:
210        der_flexi_free(list);
211        return -2;
212     LEN_ERR:
213        der_flexi_free(list);
214        return -1;
215     }
```

2.5 总结

这时我们已经拥有能够实现如PKCS或ANSI X9.62等公钥（PK）标准的所有代码了。我们必须掌握的第一件事是使用序列。没有一个通用的PK标准是直接使用ASN.1类型的（而不是序列）。

2.5.1 创建链表

本质上容器ASN.1类型是一个asn1_list类型数组。那么，让我们来研究怎样把一个简单的

容器转化为可以使用的程序代码。

```
RSAKey ::= SEQUENCE {
   N         INTEGER,
   E         INTEGER
}
```

首先，我们从定义一个链表开始。

```
asn1_list RSAKey[2];
```

然后，我们需要两个整数以保存这些值。记住，这些是C语言中的long数据类型，而不是实际上的安全RSA所要求的大数（BigNum）。这仅用于演示。

```
long N, E;
```

现在，我们必须给RSAKey中的元素进行赋值。

```
RSAKey[0].type   = ASN1_DER_INTEGER;
RSAKey[0].length = 1;
RSAKey[0].data   = &N;
```

这3行代码相当笨拙并且让ASN.1程序设计变得很痛苦。对这个问题的一种更简单的解决办法是定义一个宏，然后在C语言中用一行代码来赋值。

asn1.h:

```
#define asn1_set(list, index, Type, Length, Data) \
do {                                               \
   asn1_list     *ABC_list;                        \
   int            ABC_index;                       \
                                                   \
   ABC_list   = list;                              \
   ABC_index  = index;                             \
                                                   \
   ABC_list[ABC_index].type   = Type;              \
   ABC_list[ABC_index].length = Length;            \
   ABC_list[ABC_index].data   = Data;              \
} while (0);
```

对于那些不熟悉C预处理器的人来说，这个宏看起来很不直接。首先，我们对list和index参数分别创建一份拷贝。它们都是一些临时变量。考虑如下的调用。

```
asn1_set(mylist, i++, type, length, data);
```

如果我们一开始没有创建index的拷贝，那么对于每个使用宏的实例都会有所不同。类似地，我们也可以这样做：

```
asn1_set(mylist++, 0, type, length, data);
```

do-while循环的使用可以让我们把宏作为一个单独的C语句来使用。例如：

```
if (x > 0)
    asn1_set(mylist, x, type, length, data);
```

现在我们可以重新考虑RSAKey的定义。

```
asn1_set(RSAKey, 0, ASN1_DER_INTEGER, 1, &N);
asn1_set(RSAKey, 1, ASN1_DER_INTEGER, 1, &E);
```

这非常实用并且看起来也很舒服。

现在假设我们有一个RSA密钥，N=17*13=221，E=7，然后对其进行编码。首先，我们需要一个地方来存储密钥。

```
unsigned char output[100];
unsigned char output_length;
```

现在就可以对密钥进行编码了。我们把整个例子放在一起。

```
asn1_list     RSAKey[2];
unsigned char output[100];
unsigned char output_length;
int           err;
long          N, E;

/* Set the key  and list */
N = 221;
E = 7;
asn1_set(RSAKey, 0, ASN1_DER_INTEGER, 1, &N);
asn1_set(RSAKey, 1, ASN1_DER_INTEGER, 1, &E);

/* Encode it */
output_length = sizeof(output);
err = der_sequence_encode(&RSAKey, 2, output, &output_length);
if (err < 0) {
   printf("SEQUENCE encoding failed: %d\n", err);
   exit(EXIT_FAILURE);
}

printf("We encoded a SEQUENCE into %lu bytes\n", output_length);
```

这时，数组output[0...output_length−1]就包含了RSAKey序列的DER编码。

嵌套链表

在一个序列内部处理序列实际上是我们已经知道的东西的扩展。我们考虑如下的ASN.1结构。

```
User := SEQUENCE {
   Name        PRINTABLE STRING,
   Age         INTEGER,
   Credentials SEQUENCE {
      passwdHash  OCTET STRING
   }
}
```

像前面一样，我们创建两个链表。下面是一个完整的例子。

```
asn1_list     User[3], Credentials;
unsigned char output[100];
unsigned char output_length;
int           err;

long          Age;
unsigned char Name[MAXLEN+1], passwdHash[HASHLEN];

/* build the first list */
asn1_set(User, 0, ASN1_DER_PRINTABLESTRING, strlen(Name), Name);
asn1_set(User, 1, ASN1_DER_INTEGER,                    1, &Age);
asn1_set(User, 2, ASN1_DER_SEQUENCE,                   1, &Credentials);
```

```
/* build second list */
asn1_set(Credentials, 0, ASN1_DER_OCTETSTRING, HASHLEN, passwdHash);

/* encode it */
output_length = sizeof(output);
err = der_sequence_encode(User, 3, output, &output_length);
if (err < 0) { printf("Error encoding %d\n", err); exit(EXIT_FAILURE); }
```

在创建第一个链表时，我们向第二个链表传递一个指针（User数组中的第三方实体）。在长度域中标记相应的“1”以表示该第三方实体实际上是第二个指针。也就是说，如果Credential链表有两个选项，我们就可以在1的地方看到一对指针。

注意，连同User链表，我们只调用一次编码函数来完成整个结构的编码。编码函数将顺着指针来到第二个链表并且按照次序对其进行编码。

2.5.2 解码链表

容器的解码和编码大致相同，只不过适用的长度参数是目的缓冲区的大小。我们考虑如下的序列。

```
User ::= SEQUENCE {
   Name        PRINTABLE STRING,
   Age         INTEGER,
   Flags       BIT STRING
}
```

对于这个序列，我们需要两个unsigned char数组和一个long类型。让我们来解决它们。

```
long          Age;
unsigned char Name[MAXNAMELEN+1], Flags[MAXFLAGSLEN];
asn1_list     User[3];
```

现在我们必须设置链表。

```
asn1_set(User, 0, ASN1_DER_PRINTABLESTRING, sizeof(Name)-1,  Name);
asn1_set(User, 1, ASN1_DER_INTEGER,         1,              &Age);
asn1_set(User, 2, ASN1_DER_BITSTRING,       sizeof(Flags),   Flags);
```

注意，我们使用的是sizeof（Name）－1而不仅仅是对象的大小。这让我们有一个尾部NULL字节，以便于在解码时，C语言中的字符串函数能够发挥作用。

我们把整个例子放在一起。

```
long          Age;
unsigned char Name[MAXNAMELEN+1], Flags[MAXFLAGSLEN];
asn1_list     User[3];
int           err;

asn1_set(User, 0, ASN1_DER_PRINTABLESTRING, sizeof(Name)-1,  Name);
asn1_set(User, 1, ASN1_DER_INTEGER,         1,              &Age);
asn1_set(User, 2, ASN1_DER_BITSTRING,       sizeof(Flags),   Flags);

memset(Name, 0, sizeof(Name));
err = der_sequence_decode(input, input_len, &User, 3);
if (err < 0) {
```

```
        printf("Error decoding the sequence: %d\n", err);
        exit(EXIT_FAILURE);
    }

    printf("Decoded the sequence\n");
    printf("User Name[%lu] == [%s]\n", User[0].length, Name);
    printf("Age == %ld\n", Age);
```

这个例子假设某个称为输入的unsigned char数组已经给定并且它的长度为input_len。一旦解码成功，程序的输出会显示出用户名和年龄。例如，输出类似于下面的样子。

```
Decoded the sequence
User Name[3] == [Tom]
Age == 24
```

正如所看到的那样，*User*数组会因为像字符串类型这样的特殊类型而被修改。*User[0].length*将保存解码值的长度。注意，在我们的例子中，首先将这个数组memset为0。这可以允许我们不用管长度就可以把已解码的数组看作为一个有效的C字符串。

2.5.3 Flexi链表

正如我们在前一节讨论的那样，序列解码函数不能胜任猜测型编码。但flexi解码函数却可以让我们不用知道已编码元素的顺序和类型就可以对ASN.1数据进行解码。

虽然flexi解码函数能够胜任猜测型编码，但它不能进行猜测型分析。解码函数给我们的只有一个简单的双向链表。链表的每个节点都包含ASN.1类型，以及它们各自的长度参数和一个指向解码的数据（以独立的格式）的指针（如果合适的话）。

编码并不告诉我们当前是结构类型中的哪个元素。例如，考虑如下的序列。

```
RSAKey ::= SEQUENCE {
   D        INTEGER OPTIONAL,
   E        INTEGER
   N        INTEGER
}
```

在这个结构中，我们可以通过编码中出现的D INTEGER很容易地区分一个公共的和私有的RSA密钥。当使用flexi解码函数来处理时，我们将得到链表的两个深度。第一个深度仅包含该序列，且这个节点的孩子将是多达3项的链表（如图2-4所示）。

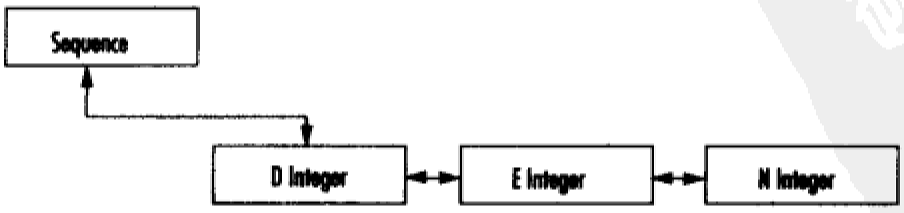


图2-4　RSAKey的Flexi解码组织

假设我们向下面那样将输出存储到MyKey中。

```
asn1_flexi *MyKey;
der_flexi_decode(keyPacket, keyPacketLen, &MyKey);
```

在这个例子中，MyKey将指向链表的序列节点。它的*next*和*prev*指针为空且还包含一个指

向*child*节点的指针。孩子节点为“D” INTEGER，如果它存在的话，它将通过*next*指针指向“E” INTEGER。

给定MyKey指针，我们用下面的代码移动到孩子节点。

```
MyKey = MyKey->child;
```

我们不必担心会丢失双亲指针，因为我们可以用下面的代码遍历回去。

```
MyKey = MyKey->parent;
```

注意，只有孩子链表的第一个入口才有双亲指针。如果我们遍历到“E” INTEGER，就不能直接移动到双亲：

```
MyKey = MyKey->child;
MyKey = MyKey->next;     /* 现在我们指向"E" */
MyKey = MyKey->parent;   /* 这是无效的 */
```

现在怎样才能知道我们是否有一个公共或者私有密钥呢？在这个特例中有一个最简单的方法就是查看孩子的数目。

```
int x = 0;
while (MyKey->next) {
      ++x;
      MyKey = MyKey->next;
}
```

如果变量*x*的值为2，它是一个公钥；否则就是私钥。这种方法非常适用于简单的OPTIONAL类型，当且仅当只有一个OPTIONAL元素。类似地，对于CHOICE修改器我们不能简单地查看其长度、类型和内容。简单的答案就是遍历链表，一旦发现区别或者能够让你确定当前看到的是什么特征时就停止。

对我们而言，幸运的是，实际应用中需要flexi解码函数的PK仅有X.590证书，它包含许多OPTIONAL组件。在稍后我们将看到，PKCS和ANSI公钥标准拥有独立的可解码序列，这些序列并不需要flexi解码函数。

2.5.4 其他提供者

本章的代码只是简单地基于LibTomCrypt项目。实际上，flexi解码函数这个想法直接取自工作于X.509认证的业界开发者。建议读者合理地使用它，因为它是一个完整的（并经过测试的）ASN.1实现，它同时也支持INTEGER、SET和SET OF类型。

2.6 常见问题

下面的常见问题，由本书的作者所回答，它们即可以用来测试你对本章所出现的概念的理解，也可以帮助你在现实生活中实现这些概念。如果希望作者解答你的问题，请浏览www.syngress.com/solutions，然后点击“Ask the Author”表单。

问：什么是ASN.1？我为什么要关注它？

答：ASN.1定义了一些可移植的方法来存储和读取各种通用数据类型，使得各种程序都可以交互操作。这有利于客户和开发者，因为它允许第三方工具解释由宿主程序产生的数据。大

部分情况下，它是对客户的一个非常有价值的卖点，因为它确保他们的数据不是专有编码体系的一部分。它避免了制造商绑定问题。

问：为什么不使用一种像XML这样的格式？

答：这个标准大概早于XML 10年就制定了，甚至那时XML只是在Tim Bray（Tim Bray, XML的作者之一）的眼睛里闪烁。尽管ASN.1在时间上早于XML，这仍不是ASN.1不选用XML的惟一原因。若把XML和ASN.1中相似的部分进行比较，会发现XML要大得多。这使得它不能很好地适应内存约束严格的硬件环境。由于许多加密是在设备上使用极少的内存完成的（智能卡是一个很好的例子），使用一种压缩的格式就显得非常有意义。有趣的是，当许多人大声疾呼需要一个分析迅速和大小减少的二进制版本XML时，一个能够完美地满足这些要求的标准早已在20世纪80年代就存在了。

问：什么标准定义了ASN.1？

答：ITU-T X.680和X.690系列标准。

问：谁使用ASN.1？

答：ASN.1是PKCS（例如#1和#7）、ANSI X9.62、X9.63和X.509系列标准的一部分。

问：哪些预编译好的加密算法库支持ASN.1？

答：OpenSSL和LibTomCrypt都支持ASN.1 DER编码和解码。本书出现的代码就是基于LibTomCrypt算法库的。建议读者在可能的时候使用它，因为它很好地集成在LibTomCrypt库的其他部分中，并且也合理地支持了INTEGER类型（也包含SET和SET OF类型）。

第3章

随机数生成

本章解决方案：

- 随机的概念
- 熵的度量
- RNG设计
- PRNG算法
- 总结

☑ 总结

☑ 快速查找解决方案

☑ 常见问题

3.1 简介

本章中，我们开始接触真正的密码学算法。首先我们研究任何现代加密体系中最重要的组件之一，通常也是很难描述的组件：随机位生成器。

许多算法因为安全性原因而依赖于能收集一些位或者数的功能，这些位或者数对于观察者来说是很难猜测或者预测的。例如，我们将看到，RSA算法需要两个素数使得公钥很难被分解。类似地，基于协议的对称分组加密算法需要随机的对称密钥，但攻击者无法获得。实际上，没有一种密码学算法不使用随机位。这是因为我们的算法都是公开的并且只有一些特殊的变量才是私有的。可以把这想象为解线性方程组：假如我想阻止你解一个有4个方程的系统，我不得不确保你最多只能知道3个互不相关的变量。虽然现代密码比一个简单的线性系统更加复杂，但其思想是一样的。

本章通篇既涉及随机位生成器也涉及随机数生成器（RNG）。它们在内容和目的上都是相同的。也就是说，任何一个位生成器都可以把它生成的位组合成一个数，任何一个数也至少可以分割成1位。

当说到随机位生成器时，我们通常是在讨论一个确定性算法，例如一个伪随机数生成器（Pseudo Random Number Generator, PRNG），它也叫做确定性随机位生成器（Derterministic Random Bit Generator, NIST, DRBG）。这些算法之所以是确定性的，是因为它们都是按照一种准确的规则集并作为软件或者硬件（FSM）算法来运行的。从字面上来说，这看起来和随机位数生成器是确定性的相当矛盾。但是，已经证明，PRNG在理论和实践上都是很实用的并且能提供真正的安全保障。正面来说，PRNG速度相当快，但这需要一些代价。它们需要某些外来的源用熵（entropy）来设置种子。即PRNG确定性状态的某些部分对外来的观察者必须是不可预测的。

这正是随机位生成器能够发生作用的关键地方。它们的惟一目的是启动一个PRNG，这样它就可以将一个固定大小的种子拉伸成一个更长的随机位字符串。通常，我们把这种拉伸称为位扩展。

随机的概念

人们已经争论真随机性的概念和存在性许多年了。某种事物是随机的（例如一个事件），是说它不能以比给定的模型所允许的更高的概率来预测其发生。曾经整个时代都在争论随机性。一些人说，某些事件如果状态没有被干扰的话是可以完全模拟的（例如光子的旋转），因此也是随机的，而另外一些人则说在某种程度上所有对事件的控制有影响的变量都可以模拟。这种争议已经被完美地消除了。量子力学已经证明在现实世界中随机性是真实存在的，并且是统治宇宙规则的关键部分（贝尔不等式给出了一个很好的理由，让我们相信没有“更深层次的”系统能够解释宇宙中能够意识到的随机性。如果感兴趣的话，可以在Wikipedia中查找相关知识）。

地板是坚固的是直接证明宇宙中存在不确定性的确切理由，这种说法并不是一种比喻。尽管相反的电荷应该相吸，但是对从来没有在原子核的表面上发现电子你是否感到奇怪。原因是因为宇宙中存在基本的不确定性。量子理论说，如果在某种程度上我能够知道电子的位置，那么我一定不能确定它的动量是多少。

当把这个原理应用于它的逻辑推理，你就能断定有相互吸引的电荷就存在稳定的原子。你现在正在读这一章，正如我们正在写这一章一样，这些事实都是因宇宙中存在不确定性而成为可能。

哇，这个相当冗长的话题让我们现在才进行到这一章的第二页。不要失望，世界上存在真实的不确定性这一事实并不妨碍我们试图用科学的和通俗的方式来理解它。

Claude Shannon在他1949发布的奠基性论文中给了我们一个能这样做的工具。在这篇论文中，他说对结果的不确定意味着它包含着尚未挖掘的信息。虽然这不同于通常的情况，但却很容易看出为什么这是真的。

假如有一个程序，它已在屏幕上打印出100万个“a”。你怎样又能确定它下一个打印出的字符就一定是“a”呢？实际上我们只能说根据所看到的，下一个字符是“a”的可能性比较大。当然，除非你看到，否则你不能确定下一个字符就一定是“a”。因此，如果又打印出一个“a”，那么它又告诉我们什么呢？这不太真实，我们已经知道仅仅是有很高的概率出现“a”，因此，“a”所包含的信息并不多。

这些论证产生了一个叫做熵或者叫事件的不确定度的概念。这个概念相当复杂，但却很容易解释。考虑投掷一枚硬币。一枚硬币有两面，考虑到硬币是相对于它的中心对称的（当在空中平稳地飞行时），所以期望它以一面落地的概率是50%。在这个事件中，熵的量为1位，它产生于等式$E=-\log_2(p)$，在这个例子中为$-\log_2(0.5)-1$。这只是相当简单的1位，这个事件中熵的数量可以通过贝尔实验室的Claude Shannon在1949年推导出的一个公式来计算。这个事件（抛硬币）有许多输出（在这个例子中是头或尾）。每个输出相应的熵等于$-\log_2(p)$[1]，其中p为产生相应输出的概率。整体的熵就是所有可能输出的熵的加权平均值。由于有两种可能的输出并且它们的熵是一样的，所以它们的平均值为$E=-\log_2(0.5)=1$位。在这个例子中，熵是表示不确定

性的另外一种方法。也就是说，如果你写下一个无限的抛硬币输出流，那么这个理论告诉我们平均每次投掷只需要1位就可以表示了。建议读者阅读一下Arithmetic Encoding（算术编码）以了解是怎样对二进制字符串进行压缩编码的。

但是，对于投掷硬币，我们假设存在一些无法预测或者模拟的影响。在这个例子中，这样做实际上是不正确的。考虑到平均的旋转速度和高度，硬币投掷的概率会高于50%。这并不意味着硬币投掷会产生零阶熵，实际上，甚至对一个半精确模型来说，每次投掷似乎也近似只需要1位。这里的关键概念是如何模拟熵。

那么，什么是“熵”？人们也许已经写了很多关于这个问题的基本论文。作为科学家，理解它的最好的办法就是从计算机科学的角度来看。一串数字，如果没有比它更短的可以描述它的形式的程序，那么它就是随机的。

这叫做Kolmogorov复杂度，它的深入讨论已经超出了本书的范围。感兴趣的读者可以参阅G.J.Chaitin所写的书《*The Quest for Omega*》。这本书可以从下面的网址获得www.arxiv.org/abs/math.HO/0404335。

在我们的讨论范围之内将如何创建一个非确定性（从软件的角度来看）随机位生成器，这个我们会在后面的章节中讨论。首先，我们不得不确定怎样在一个事件流中对熵进行观察。

3.2 熵的度量

在开始构造一个随机位生成器之前，我们要做的第一件事就是，找到一种或几种方法来估算连同每个生成的位一起，我们实际共生成了多少熵。这让我们不得不谨慎地使用RNG来初始化一个PRNG，并假想它在状态中能够生成某个最小数量的熵。

要清楚地看到，我们说的是估算而不是确定。有许多已知有用的RNG测试，但是却没有一个简短的用于模拟一个给定随机位子集的方法列表。确切地说，对于任何一个长度为L位的位序列，你不得不测试所有的长度为L－1位或者更少的生成器来检验它们是否可以生成这个序列。如果它们可以，那么这个L位的位序列实际上只有L－1（或更少）位的熵。

实际上，许多RNG测试甚至并不计算熵，至少不是直接地。相反，许多测试只是一些简单的模拟，这叫做Monte Carlo模拟。他们执行一个被很好地研究了的模拟，这个模拟在关键决策点使用RNG输出来改变模拟输出。为了辅助模拟，还使用了预测器，它观察RNG的输出位并使用这些输出位来模拟相应的事件。

预测一个PRNG的输出通常比一个分组密码或者散列算法做同样的事情要容易得多。PRNG比分组密码使用更少的字节操作，这个事实意味着实际上很少的字节操作使得信息更加安全。简单来说，从它自身来讲，这正是弱点所在，但这也正是它理解起来比较直观的原因。由于PRNG常用来初始化其他的密码学原件，例如分组密码或者消息认证码，所以值得一提的是，攻破PRNG的人也有可能攻破系统剩下的部分。

各种各样的程序，例如DIEHARD和ENT已经在网上存在数年了（DIEHARD尤其流行——www.arxiv.org/abs/math.HO/0404335），它们包含了各种模拟和预测器。虽然它们通常擅长于指出设计上的缺陷，但它们不擅长于指出是否是好的算法。也就是说，即使一个算法通过了其中一个或者所有的测试，它也仍然有可能是有缺陷的，用于密码学目的也是不安全的。即我们不

应该完全地忽视它们，并且要开发一些我们能够应用的基本测试。

在设计一个RNG时，一些关键测试很有用。它们绝不是一个测试的详尽列表，但足够 我们用来快速的筛选出RNG。

3.2.1 位计数

“位计数”测试只是简单的计算0和1的个数。在理想的情况下，我们期望的是一个位于大量位数据流上的均匀分布。这种测试可以马上排除偏差值超过1位或者其他的RNG。

3.2.2 字计数

类似于位计数，这种测试计算k位字的个数。通常这可用于k的值为2～16时的情况。对于一个足够大的数据流，你可以看到任何一个k位字都是以$1/2^k$的概率出现。这种测试可以排除RNG中的抖动。例如，数据流01010101……可以通过位计数测试但不能通过$k=2$时的字计数测试。数据流001110001110……即可以通过位计数测试也可能通过$k=2$时的字计数测试，但当$k=3$时就不能通过。

3.2.3 间隙计数

这种测试查看值为0的位之间（或者值为1的位，这取决于你想查看哪一种）的间隙大小。例如字符串00的间隙为0，010的间隙为1，等等。对于一个足够大的数据流，期望k位大小的间隙以$1/2^{k+1}$的概率出现。设计这种测试是用来捕捉在时钟周期结束以后便产生一个给定值的RNG（即使是一个很短的周期）。

3.2.4 自相关测试

这种测试用来确定一个位子集是否和同一个字符串中的另一个位子集相关。虽然它形式上是针对连续的数据流而定义的，但它也可以很好地应用于有限离散信号。下面的等式定义了自相关（autocorrelation）。

$$R(j) = \sum_n x_n x_{n-j}$$

其中x是被观察的信号，R(j)是对延迟j的自相关系数。在继续进行下去之前，我们先来看这部分中将要用到的几个术语。当两个项比看起来不同更相似（相关）就称它们是相关的。例如，字符串1111和1110是相似的，但是照那样看来，1111和0000也是相关的，因为第二个确实是第一个的相反值。如果两个项有相同数量的差别和相似性，就称它们是不相关的。例如字符串1100和1010就是不相关的。

在离散量的世界中，是以位为单位进行采样的，采样的取值范围为{0,1}，在应用变换之前首先要把它们映射成{−1,1}。如果不这样做，对于相关采样（互为相反数的），自相关函数的结果将趋于0。

还有一种更加友好的解决方案就是计算两个数的异或（XOR）之和。新的自相关函数如下。

$$R(j) = \sum_n x_n \text{ xOR } x_{n-j}$$

对不相关的数据流，结果将趋向于$n/2$，对于相关的数据流，结果将趋向于0或n。如果把这个新自相关函数用于有限的数据流中，将变得复杂起来。采样值小于0的索引是什么？典型的解决办法是从j开始向上求和并计算出一个接近（$n-j$）/2的值。

autocorrelate.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
void printauto(int *bits, int size, int maxj)
{
   int x, j, sum;
   for (j = 1; j <= maxj; j++) {
      for (x = j, sum = 0; x < size; x++) {
         sum += (bits[x] ^ bits[x-j]);
      }
      printf("Lag[%4d] = %5d (expected %5d)\n",
              j, sum, (size - j)/2);
   }
}
```

这个函数输出一个采样数组的自相关系数，当到达最大延迟时结束输出。我们来看一个简单的偏向PRNG。

```
l = 0;
for (x = 0; x < SIZE; x++) {
   if (rand()&1) {
     bits[x] = 1;
   } else {
     l = bits[x] = rand()&1;
   }
}
```

这个生成器将以50%的概率输出最后一位，否则将输出一个随机位。我们通过自相关测试来验证下面这些数据。

```
Lag[   1] = 262638 (expected 524287)
Lag[   2] = 393784 (expected 524287)
Lag[   3] = 459840 (expected 524286)
Lag[   4] = 492175 (expected 524286)
Lag[   5] = 508232 (expected 524285)
Lag[   6] = 515917 (expected 524285)
Lag[   7] = 520401 (expected 524284)
Lag[   8] = 521771 (expected 524284)
```

这是从多于1 048 576次采样（2^{20}）中取的一部分数据。我们可以从列表中的前8个延迟看出这远达不到理想的情形。现在只考虑rand() & 1测试。

```
Lag[   1] = 524999 (expected 524287)
Lag[   2] = 525284 (expected 524287)
Lag[   3] = 524638 (expected 524286)
Lag[   4] = 525564 (expected 524286)
Lag[   5] = 524480 (expected 524285)
Lag[   6] = 523917 (expected 524285)
Lag[   7] = 524692 (expected 524284)
Lag[   8] = 524081 (expected 524284)
```

正如我们看到的那样，相关值比前面的更接近于期望值。我们可以从这个实验中得出一个结论，前者是一个不好的位生成器，而后者更加合适。要记住的是，这个测试并不意味着后者可以作为一个理想的RNG来使用。取自glibc中的rand()函数并不是一个安全的PRNG，而且它的内部状态非常小。这是一种常见的错误，请您不要犯这样的错误。

这种通俗的实现有一个弊端，由于它需要一个很大的缓冲区因而运行效率不高。特别是在RNG运行的时候，可以执行这个测试来确保RNG不会产生相关的输出。已证明了窗口化相关测试的实现也是相当简单的。

wincor.c:

```c
/* windowed autocorrelation */
#define MAXLAG 16
int window[MAXLAG], correlation[MAXLAG];

void wincor_add_bit(int bit)
{
   int x;
   /* compute lags */
   for (x = 0; x < MAXLAG; x++) {
      correlation[x] += window[x] ^ bit;
   }

   /* shift */
   for (x = 0; x < MAXLAG-1; x++) {
      window[x] = window[x + 1];
   }
   window[MAXLAG-1] = bit;
}
```

在这段代码中，我们定义了16个延迟变量（MAXLAG），并且在整个自相关函数中这个函数一次只处理1位。它并不输出一个结果，而是只修改*correlation*数组这个全局变量。这个函数实质上是一个带有并行求和操作的移位寄存器。它可以很好地适用于硬件实现，也适于SIMD软件实现（尤其是带有更长的延迟值）。这个测试只有在处理长的位数据流时才能发挥最好的作用。默认窗口会用值为0的位进行填充，这些位可能会和那些正在添加的位相关，也可能不相关。一般来说，对于自相关测试，应避免将其用于字符串的次数比用于至少数千位的数据流的次数还要少。

3.3 它能有多糟

在前面提到的所有测试中，你似乎已经知道了“正确的”答案，但是你不会总是能得到正确的答案。例如，如果你完美地投掷了20次硬币，你期望有10次是正面朝上的。但如果实际有11次或者9次正面朝上会怎么样？这意味着这枚硬币是偏向于一面的吗？实际上，20次投掷就有2^{20}（1 048 576）种可能性（或者近似等于），在这所有的可能中，仅有184 756次，即大约18%的可能情况下有10个是正面朝上的。

统计数据能告诉我们输出位是多么特殊的一个集合，而这是建立在假设生成器是好的基础之上的。但是在实际应用中，通常是有一些常识就足够了。然而，甚至只有8个正面朝上也是可能的，但只有两个的情况的确是无法预知的。如果20个都正面朝下，那么这种情况肯定能说

服你这是一个两面的硬币。用于测试的数据越多，你的直觉就越可靠。

3.4 RNG设计

实际应用中的大部分RNG都是围绕着某些事件收集处理机制进行设计的，如图3-1中的流程所示。RNG设计并没有真正的标准，因为标准机构更趋向于确定性的一面（例如PRNG）。一些经典的RNG，例如Yarrow和Fortuna设计（稍后将进行讨论），都是很灵活的，而且是由这个领域中一些最优秀的密码学家所设计的。但是，我们真正想要的是一个简单且有效的RNG。

针对这个任务，我们将从Linux内核RNG中获取经验。它是一个非常好的模型，因为在它集成于内核内部的同时也暴露给用户空间中的外部实体。正因为如此，它不得不规范地使用栈、堆和CPU时间等资源。

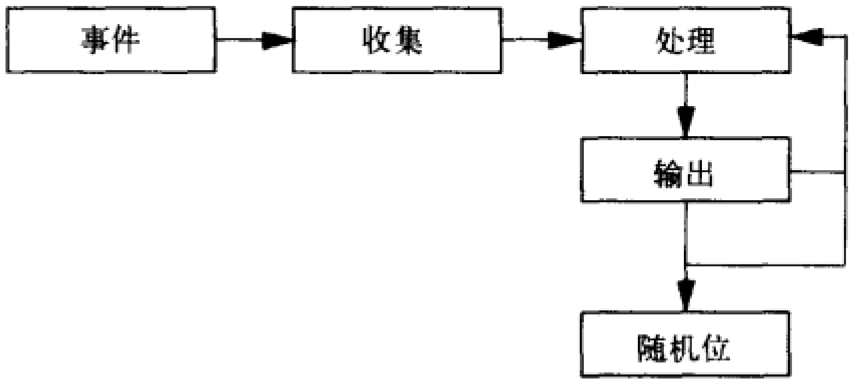


图3-1 RNG流程图

典型的RNG工作流程可以分成如下几个离散的步骤：事件、收集、处理和输出反馈。并不是所有的RNG都遵守这个全面的设计概念，但它很常用。

对此有些误解的人第一个想法就是，认为RNG是依靠某些像中断定时器一样的随机源并直接返回输出的。在现实生活中，至少从信息理论的观点来看，一些熵源是不够理想的。RNG的目标是从输入事件（种子数据）中提取熵，并且它只返回和已初始化的熵的位数同样多的随机位给调用函数。实际上这是实际应用中RNG和PRNG的惟一关键区别。

RNG和PRNG的另外一个不同的地方是RNG是直接依赖于种子的熵的。PRNG意味着只初始化一次（或者很少），然后输出比它内部状态中的熵更多的位。这个关键区别限制了RNG的使用条件，比如很难取得种子事件或者种子事件的熵很低。

3.4.1 RNG事件

如果RNG试图从某个事件中提取熵，那么这个事件就是一个RNG事件。它们是系统中不以高度可预测的时间间隔或者状态而发生的事件。RNG的目的就是捕捉这个事件，收集可用的熵并将它传递给RNG内部状态进行处理。

事件以各种形态和大小发生。一个RNG使用什么样的事件高度取决于正在开发的是什么平台。在桌面或服务器平台上，第一个“长得低的果子”是硬件中断，例如那些由键盘、鼠标、定时器（漂移）、网络和存储设备引起的中断。这些中断的全部或者部分都以各种各样的形式存在于大部分的桌面和服务器平台上。甚至在带有网络连接的嵌入式平台中，它们也常存在。

其他的典型来源有定时器偏差、模数转换噪声和二极管泄漏。我们首先来考虑桌面和服务器平台。

1. 硬件中断

事实上，在所有带有硬件中断的平台上，触发中断的进程都是非常一致的。在能断言中断的设备里，它们发出一个中断信号（通常是一条专用引线），控制器（例如可编程中断控制器(PIC)）可以对这个信号进行检测，列入优先级队列然后给处理器发信号。一旦处理器检测到中断，就停止处理当前的任务，转入内核处理模块，然后由它来响应中断并处理中断事件。在处理完中断事件之后，中断处理程序将控制权返回给中断的任务，接着任务的处理程序继续进行下去。

我们将在中断处理程序内部从事件中观察和提取熵。这个额外的步骤产生了一个问题，或者对大部分的开发者来讲它应该产生一个问题，因为中断处理程序的延迟时间必须足够小以确保系统的性能。如果一个1kHz的定时器中断需要10 000个时钟周期来处理，那么它将从用户的任务中每秒偷取10 000 000个时钟周期（即让你的处理器的处理速度下降了10M Hz)。中断不得不以很快的速度进行，这意味着为了“收集”熵而所能做的就很有限了，也许就只能是一个内存复制操作。这也正是为什么在RNG构造时会有明显的收集和处理步骤的原因所在。通常是在从用户任务中的RNG做一个读操作之后，处理才会进行。

我们能够从中断处理程序中收集到的第一块熵就是来自于定时器中断，它的发生时机不容易确定，它可能发生在两个不同的中断之间，特别是在系统加载的过程中它也经常发生。注意，定时器在RNG构造中的使用很复杂，我们将在后面进行简短的介绍。

第二块熵就是和事件相关联的数据。例如，对于一个键盘中断，它是那个按下的键；对于一个网络中断，它是帧；对于一个鼠标中断，它是坐标或者输入流，等等（显然，在系统加载的时候复制全部的帧是一个很严重的性能瓶颈，这时不得不需要某些级别的过滤处理)。

最后一块有用的熵是对一个具有很高的分辨率的自由运行计数器的捕捉。对于那些负载不够多，以致于对中断的组合并不能得到足够的熵的系统，这极其有用。例如，一个网络中断的发生就可能没有很多的熵。中断发生时的精确时钟分段信号是不好确定的，因此它们也具有熵。当时钟是自由执行并且没有和任务调度或者中断调度进程绑定时，这特别实用。在x86处理器中，RDTSC指令可以用于这种处理，因为它实质上是一个非常高精度的自由运行计数器，它不受总线时间、设备时间或者甚至是处理器指令流的影响。这种熵的最后来源是定时器偏差的一种形式，我们将在下一个部分中进行介绍。

警告 对于密码学应用的目的来说，在事件中包含一个高分辨率的定时器以增加收集到的熵是非常有用的。但是，在某些平台上，捕捉定时器会给进程增加严重的指令周期损耗。

在x86 Intel和AMD平台上，RDTSC指令需要12～100个时钟周期（这取决于模块和处理器的状态)。RDTSC也是一个串行化操作，意思是说在读计数器之前，处理器必须首先退回所有的操作码。在大部分的处理器中，它也类似于一个原子操作，也就是说在执行时处理器不能被中断。

总体来看，根据中断的频率，花费100个时钟周期并不可怕。键盘中断大约1min

触发300次。对2GHz的CPU来说，这意味着在CPU时间的每秒中，用户都用花去0.00025ms来调用RDTSC指令。

提示 在某些平台上，例如G3和G4苹果电脑中的PowerPC，CPU定时器实际上远离总线定时器的再分频器。所有其他和FSB连接的以及远离北桥的设备都使用总线定时器。在这些平台中，把CPU定时器和中断绑定在一起不是一种不好的想法，尤其也不是一种好的办法。也就是说，它不会造成损害，但也不会有所帮助。

所有的这3块信息每一个只有很少的熵，但是将它们结合在一起就会有足够多的非零熵，这样就能够进行收集和进一步处理。

2. 定时器偏差

当两个或多个电路不是锁相电路的时候，定时器偏差就会发生。它们可能有相同的振荡频率，但是受许多因素的影响，例如电压和温度，一个周期的起始和结束可能会有所偏移。在数字设备中，它常用来减少非同步时序（distinct clocks）、PLL设备和自同步时序（self-clocking）（例如HyperTransport总线）的使用。

对于软件来说，通常很难产生不同步且是一个接一个地锁在一起的定时器。例如，PCI时钟应该和PCI总线上的每一台设备都同步。它也应该锁定连接在PCI控制器上的南桥，等等。

在x86世界中，已证实至少有一种足够有用（虽然很慢）的来源。但还是使用了RDTSC指令。至少对于Intel和AMD的处理器定时器来说，它并不是直接和PIT或者ACPI定时器绑定。因此可以用一个相当简单的循环来提取熵。

timer_bit.c:

```
#include <signal.h>
#include <stdio.h>

volatile int x, quit, capture;
void sighandle(int signo) { capture = x; quit = 1; }
int main(void)
{
   int y;
   signal(SIGALRM, sighandle);
   for (y = 0; y < 16; y++) {
      quit = 0;
      alarm(1);
      while (!quit) { x ^= 1; }
      printf("%d", capture);
      fflush(stdout);
   }
   printf("\n");
   return 0;
}
```

上面这个例子显示出16位之后就返回到外壳程序。在这个例子中，变量x和1的异或（XOR）操作在0和1之间产生了一个高频度的时钟振荡，这与处理器解码、调度、执行以及退回操作的速度一样快。对于一个给定的1s周期来说，这个时钟的精确频率依赖于很多因素，包括缓存的内容、其他中断、优先于该任务的其他任务以及当alarm信号发生时，处理器运行到了循环中的哪个地方。

在x86平台中，这个while循环实质上类似于下面的一段汇编代码。

```
top:
   mov eax, [x]
   xor eax, 1
   mov [x], eax
   mov eax, [quit]
   test eax,eax
   jz top
```

一开始可能看起来很奇怪，当一个带有内存操作数的XOR就足够的情况下，编译器却为XOR操作生成了3个操作码。但是，"x^=1"这个操作实际上定义了3个原子操作，因为*x*是可变的，这3个分别是一个加载操作、一个XOR操作和一个存储操作。从程序设计的角度来看，在退出循环的时候，即使只有一个信号整个XOR语句也会执行。这也正是为什么信号处理程序能够在设置*quit*标志之前就能够捕捉到*x*值的原因。

中断（或者在本例中为一个信号）能够在任何指令之前终止程序的运行。它们当中只有一个能够修改内存中*x*的值，同时信号处理程序也是可以访问这个内存的。图3-2给出了在Opteron工作站上产生的一些采样输出。

```
0010001010111010
0101000110111100
1000101110111111
```

图3-2 timer_bit函数的采样输出

如果继续进行下去，我们可能会看到具有某些偏差的形式，但是至少熵不为0。但是还存在一个问题，就是这个生成器的速度很慢。它每秒只取1位并且每个输出的熵可能都不接近1位。这意味着对于一个128位的序列，这将需要1 280s或者大约22min。

一种解决的办法是在警报信号上减少延迟的长度。但是，如果我们在非常短的一个周期内进行采样，频率偏移的可能性将会降低，这也会降低输出中所包含的熵。也就是说，很可能两个自由运行计数器在很短的周期时间内仍然是同步的，但长时间的周期就不是了。

如何合适地调整延迟已经超出了本书的范围。每一种平台，甚至在x86的世界中，每一种实现都是不同的。电源供应、位置和板载组件的质量都会影响在时钟周期之间锁相将会损失多少以及多久。

对于硬件来说，这些条件都可以为获得好处而重造，但是很难把它们融合在一起。由同一个电源（或者电缆）供电的两个时钟，将释放出相同的电压，这证明了它们具有相同的设计以及功率曲线。实际上，你并不希望它们是锁相的，而且也不希望它们产生很大的漂移，当然这只是在很短的时间周期内。没有两个时钟是完全一样的，这是一个事实，但是，对于用在RNG上的目的来说，它们可能会足够相似，以至于它们的状态中的任何不确定性都不能发挥什么作用。硬件时钟必须通过两个不同的电缆线来供电，例如两个不同的电池。这种独立性使得很难在成本基础固定的情况下将它们融合在一起。

这种熵收集机制的一种最实用的形式是，使用带有一个具有高分辨率的自由运行计数器的中断。甚至，正常的系统定时器调度中断都应当通过这个高精度的定时器来收集。

3. 模数转换（Analogue to Digital）错误

模数转换器（ADC）将捕捉到的输入波形信号进行量化和数字化转换。其结果通常是一个脉冲编码调制流（Pulse Code Modulation stream），它使用一群位来表示离散时刻中的信号强度。这常见于作为麦克风和线输入组件来使用的声卡中，以及电视调谐解码器上和无线电设备中。

正如在定时器偏差中所说的一样，我们试图从电路中找出其他不同的可以感知的缺陷以提取熵。最明显的来源是采样的最低位，它可以持续稳定的以一种值或其他值出现，这取决于采样什么时候进行锁存、传给ADC的相对电压以及其他环境噪声。实际上，定时器偏差（当锁存采样的时候发出信号）可以给已得到的采样增加熵。

作为一个实验，考虑播放一盘CD并坐在一个合适的隔离的音响工作室中使用麦克风将输出录制下来。两个位数据流不是完全吻合的可能性非常高。它们之间的跨相关（cross-correlation）非常强，尤其是在设备的质量很高的时候。但是，即使是在最好的环境下，仍然只能得到有限的熵。

另外一个更容易完成的实验是在你的桌面上进行录制，使用（或不使用）一个带有机器所放地方的环境噪声的麦克风。即使不使用麦克风，在一个Tyan 2877上声音编解码器也能检测出通过ADC的噪声是什么级别的。虽然这些采样非常小，但是即使在如此极端的环境下仍然是有熵可以收集的。理想情况下，如果能带有一个录制设备，例如一个麦克风就最好不过了，这样也可以捕捉环境噪声。

在实际应用中，ADC很难用于桌面和服务器环境。它们更有可能用于自定义的嵌入式设计中，使得对这些组件的使用不致于影响用户。例如，如果你的系统正开始捕捉视频，而此时你正想要使用声卡，这就让人很讨厌了。从我们希望收集的数据来看，它仅仅是最低位，这大大减少了收集处理中的存储空间需求。

3.4.2 RNG数据收集

既然我们知道了可以从一些源中得到熵，那么我们就必须设计一种能够快速收集它们的办法。收集处理这一步的目标是为了减少事件的延迟时间，这样中断就可以在一个合理的可控的时间内运行。

实际上，当没有响应中断的时候，收集层会把收集到的熵传递给处理层。当我们不得不使用这些熵来做一些有意义的事情时，我们不必马上将这些步骤都删除。

逻辑上的第一步是拥有一块预分配好的内存，我们可以把转储出来的数据存在它里面。但是，这又导致了一个新的问题。内存块的大小是固定的，这意味着当块满了以后，要么忽略新的输入，要么把已满的块送去处理。简单地丢掉新的（或者更旧的）事件来确保缓冲区没有溢出并不是一种合适的选择。

一个知道熵可以被丢弃的攻击者可能会触发一系列的低熵事件来确保收集到的数据的质量很低。Linux内核通过使用一个两级处理算法来解决这个问题。第一步，它们把熵和一个可以保留熵的线性反馈移位寄存器（Linear Feedback Shift Register, LFSR）进行混合。一个LFSR实际上是一个PRNG装置，它把内部状态的线性组合作为输出（为什么使用LFSR？假设我们对设备中出来的位进行XOR，那么就会有一个偏值。这个偏值将在转储区域收集不同的位。虽然LFSR并不完美，但它却是一种可以使偏值在内存范围内进行传播的快速方法）。在Linux内核中，与使用自身的线性组合来修改LFSR内部状态不同的是，它们将内部的位和事件中收集到的位相异或（XOR）。

异或操作特别适用于这种情形，因为这就是已知的熵保留，或者简单地说，是一个平衡操

作。很显然，你的熵不能超过内部状态的大小。但是，如果状态的熵已满，那么不管输入的是什么，它都不能被减少。例如，考虑一次一密（One Time Pad, OTP）。在一个一次一密中，即使明文的熵很低，例如英语文本，输出的密文仍然是每位有一位的熵。

1. LFSR的基础知识

在这一步中，LFSR仍然十分有用，因为它们可以很快地实现。基本的LFSR是由一个L位的寄存器组成，它移动一次并将移出的位和移位寄存器中选择的位进行异或。这些被选择的位称为“tap位”。例如：

```
unsigned long clock_lfsr(unsigned long state)
{
   return (state >> 1) ^ ((state & 1) ? 0x800000C5 : 0x00);
}
```

这个函数提供了一个32位的LFSR，它的tap类型为0x800000C5（位31、7、6、2和0）。现在，在RNG中进行数据移动只需要简单地向下面这样做。

```
unsigned long feed_lfsr(unsigned long state, int seed_bit)
{
   state ^= seed_bit;
   return clock_lfsr(state);
}
```

收集块中的每个位都会调用这个函数，直到收集块为空为止。因为LFSR是熵保留的，不管提供多少种子位，这个结构最多在状态中产生32位的熵，但是这远远不能满足任何密码学要求，它必须被加到一个更大的缓冲池中去。

2. 基于表的LFSR

当增加一些字节时，用一次一位的时钟控制LFSR是一种很慢的操作。作为中断来运行，它是非常慢的。已经证明，我们不需要用一次一位来控制LFSR。实际上，可以让它们一次处理任何位数，特别是当使用查找表（lookup tables）时，不需要分支（LSB测试）就可以完成整个操作了。严格地说，我们可以在不同的域上使用LFSR，例如$GF(p^k)^m[x]$形式的扩展域。但是，这超出了本书的范围，所以我们应该慎重地避免它们。

最有用的时钟控制数量是8位，这样我们可以一次合成一个字节的种子数据。它也保持了表的大小为1KB。

lfsr32.c:

```
static unsigned long shiftab[256];
unsigned long step_one(unsigned long state)
{
   return (state >> 1) ^ ((state & 1) ? 0x800000C5 : 0x00);
}
```

这是我们熟悉的32位步长函数，它只控制LFSR一次。

```
void make_tab(void)
{
   unsigned long x, y, state;

   /* step through all 8-bit sequences */
```

```
012      for (x = 0; x < 256; x++) {
013         state = x;
014         /* clock it 8 times */
015         for (y = 0; y < 8; y++) {
016            state = step_one(state);
017         }
018         /* store it */
019         shiftab[x] = state;
020      }
021   }
```

这个函数创建了一个包含有256个入口的表shiftab。我们处理所有的256个低8位并控制寄存器8次。这样做得到的最终结果（第19行代码）就是将要与寄存器进行异或的数据。

```
023   /* clock the LFSR 8 times */
024   unsigned long step_eight(unsigned long state)
025   {
026      return (state >> 8) ^ shiftab[state & 0xFF];
027   }
028
029   /* seed 8 bits of entropy at once */
030   unsigned long feed_eight(unsigned long state,
031                            unsigned char seed)
032   {
033      state ^= seed;
034      return step_eight(state);
035   }
```

第一个函数（第24行代码）步入LFSR 8次，每次都只是一个移位操作、一个查表操作和一个异或操作。在实际应用中，它比8次还要快，因为我们从代码中移去了条件异或操作。

第二个函数（第30行代码）在一个单独的函数调用中使用8个位来初始化LFSR。它正好等价于使用一个单步函数来提供1个位（从最低位到最低位）。

```
037   #include <stdio.h>
038   #include <stdlib.h>
039   #include <time.h>
040   int main(void)
041   {
042      unsigned long x, state, v;
043
044      make_tab();
045      srand(time(NULL));
046      v = rand();
047
048      state = v;
049      for (x = 0; x < 8; x++) state = step_one(state);
050      printf("%08lx stepped eight times: %08lx\n", v, state);
051
052      state = step_eight(v);
053      printf("%08lx stepped eight times: %08lx\n", v, state);
054      return 0;
055   }
```

如果你不相信这个技巧是有效的，那么可以考虑多次运行这个演示程序。既然有了一种更有效地修改LFSR的办法，我们就能继续让它的规模变得更大。

3. 大型LFSR的实现

从实现的角度来看，理想情况下，你希望LFSR具有如下两种性质：简单以及包含很少的tap。LFSR越简单，移位的速度就越快并且需要的表就越小。另外，如果tap很少，大部分的表会包含值为0的位并且我们可以对它进行压缩存储。

但是，从安全性的角度来看我们的希望却恰恰相反。LFSR越大包含的熵就越多，tap的数量越多熵在寄存器里的传播就越充分。密码学家的任务就是知道该怎样以及在哪里对实现效率和安全性划定界限。

在Linux内核中，他们使用一个大型的LFSR，在将它发送出去以进行加密处理之前对熵进行混合。这产生了一些很占用空间并且污染了处理器的数据缓存的缓冲池。我们解决这个问题的办法是很简单的并且也很有效。与使用一个大型的LFSR不同的是，我们仍然坚持使用32位的LFSR并且周期的将它传送给处理层。把LFSR种子添加到处理缓冲池的实际步骤将尽可能地简单以保持低的延迟时间。

3.4.3 RNG处理和输出

RNG处理步骤的目的是，取用种子数据并将其转化为某些可以返回给调用函数的东西，而不影响到RNG的内部状态。在这个时候，我们仅仅是线性地将输入的熵混合到处理模块中。如果我们简单地用一个调用函数来处理的话，它们就能够为LFSR解决线性方程并且能够知道内核当中正在发生什么事件（严格来说，如果缓冲池的熵就是缓冲池的大小，这也不算是一个问题。但是，在实际应用中，并不是总是会这样）。

在处理层中所使用的一般技巧是，使用一个单向散列函数来处理种子数据并将它“搅拌”到一个可以从中得到熵的RNG状态中。在我们的构造函数中，使用了SHA-256散列函数，因为它相当大并且相当有效。

处理层的第一部分是混合来自收集层的种子熵。我们注意到SHA-256的输出实际上是8个32位字，因此，处理缓冲池也是256位。我们将使用循环处理的办法把32位的LFSR种子异或为状态中8个字的其中之一。在把状态进行搅拌以产生输出之前，我们必须从收集层中至少收集8个字（可能会有更多）。

实际上搅拌数据相当简单。我们首先把搅拌函数的调用次数和状态中的第一个字相异或。这防止当RNG用于非阻塞模式（nonblocking mode）（稍后我们将详细介绍）时会转变成固定点。接着，我们添加当前RNG缓冲池中的前23个字节，并且对这个55个字节长的字符串进行散列以形成新的256位熵缓冲池，调用函数可以从中读取以得到随机字节。然后把散列输出和该256位的收集状态进行异或以防止回溯攻击。

关于这样做的第一件奇怪的事就是搅拌计数器的使用。RNG工作的一种模式是“非阻塞模式”。在这种模式中，我们所做的更像是一个PRNG而不是RNG。当缓冲池为空时，我们并不等待来自收集层的另外的8个字；我们只是简单地重新搅拌当前的状态、缓冲池以及搅拌计数器。该计数器确保每次搅拌时散列的输入都是惟一的，以防止产生固定点和短周期（当散列的输出等于散列的输入时，固定点就会产生。当发现一个碰撞时，短周期就会产生。这两种情况都不太可能发生，但是为避免它们所花的代价却是如此得微不足道，因此这是一种很好的安全

措施)。

另外一个奇怪的事情是我们只使用缓冲池中的23个字节，而并不是所有的32个。在理论上，我们并没有不能使用所有的32个字节的理由。相对于其他原因而言，这个选择更有可能是因为性能上的考虑。SHA-256一次操作64个字节的分组（参见第5章散列函数以获得更多的细节）。散列函数总是以一个0x80字节加上消息长度的64位编码对正在进行散列的消息进行填充。这就是所谓的MD强化（MD-Strengthening）技术。我们所关注的仅仅是长度。假如我们用满了全部64个字节，那么消息将被填充9个字节，并且需要两个分组（64个字节的）来进行散列以产生输出。通过使用状态中的32个字节和缓冲池中的23个字节，填充的9个字节正好组成一个分组，这加倍提高了性能。

你也许会好奇为什么我们包含了前一个输出的所有字节。毕竟，它们潜在地给了攻击者一个观察的机会。其原因大概是因为实际比理论更加安全。大部分的RNG会被存入私有缓冲区，用来进行像RSA密钥生成或者对称密钥选择之类的事情。其输出仍然能够含有熵。包含前一个输出的所有字节的这个操作也是很自由的，因为SHA-256散列函数会将这23个字节填充为0。其实，这没有什么不妥，甚至在某些环境下还会有所帮助。毫无疑问，你仍然需要一个新的熵源来安全地使用这个RNG。重复利用散列的输出不会添加熵，它仅仅是为了防止降级。

我们现在对上面的进行总结：

rng.c:

```
/* our SHA256 function we need */
void sha256_memory(const unsigned char *in, unsigned long len,
                          unsigned char *out);

/* the LFSR table */
static const unsigned long shiftab[256] = {
   0x00000000, 0x1700001c, 0x2e000038, 0x39000024, 0x5c000070,
   0x4b00006c, 0x72000048, 0x65000054, 0xb80000e0, 0xaf0000fc,
   0x960000d8, 0x810000c4, 0xe4000090, 0xf300008c, 0xca0000a8,
   0xdd0000b4, 0x7000004b, 0x67000057, 0x5e000073, 0x4900006f,
<snip>
   0x9b0000d3, 0xa20000f7, 0xb50000eb, 0x6800005f, 0x7f000043,
   0x46000067, 0x5100007b, 0x3400002f, 0x23000033, 0x1a000017,
   0x0d00000b
};
```

这是32位LFSR中tap为0x800000C5所对应的表。我们对这个列表进行剪裁以节约空间。完整的列表可以在网上获得。

```
/* portably load and store 32-bit quantities as bytes */
#define STORE32L(x, y)                                  \
     { (y)[3] = (unsigned char)(((x)>>24)&255);   \
       (y)[2] = (unsigned char)(((x)>>16)&255);   \
       (y)[1] = (unsigned char)(((x)>>8)&255);    \
       (y)[0] = (unsigned char)((x)&255); }

#define LOAD32L(x, y)                                   \
     { x = ((unsigned long)((y)[3] & 255)<<24) | \
           ((unsigned long)((y)[2] & 255)<<16) | \
           ((unsigned long)((y)[1] & 255)<<8)  | \
           ((unsigned long)((y)[0] & 255)); }
```

这两个宏对我们而言是新的，它们将会在后面的章节中越来越多地使用。这些宏以可移植的形式存储和加载32位的little endian格式的数据（分别地）。这可以让我们避免平台之间的兼容性问题。

```
074  /* our RNG state */
075  static unsigned long LFSR,      state[8],    word_count,
076                       bit_count, churn_count;
```

这是RNG内部状态。*LFSR*是当前正在累积的32位字。*state*是形成熵的收集缓冲池的32位字的数组。*word_count*计算由*LFSR*添加给*state*的字数。*bit_count*计算添加给*LFSR*的位数，*churn_count*计算搅拌函数被调用的次数。

*bit_count*变量是用.4固定点编码值来解释的。意思是说，我们将整数分成两部分：一个28位的整数部分（这是在32位平台上，如果是在64位平台上将为60位）和一个4位的小数部分。*bit_count*的值照字面意思理解等于下面使用C语言语法的表达式（bit_count >> 4） + (bit_count/16.0)。这使得我们可以把熵的位的小数部分加到缓冲池里去。

例如，当发生一个鼠标中断时，我们把X,Y按钮和滚动位置加到RNG中。我们可以说它们一共提供给RNG 1位的熵或者说每个采样提供0.25位的熵。因此，我们把0.25 * 16 = 4作为熵的数量。

```
078  /* pool the RNG data comes out of */
079  static unsigned char pool[32];
080  static unsigned long pool_len, pool_idx;
```

这是来自RNG的缓冲池状态。它包含将要返回给调用函数的数据。*pool*数组保存最多32个字节的输出，*pool_len*表示还剩下多少字节，*poo_idx*表示将要从数组中读入的下一个字节。

```
082  /* add a byte of entropy to the RNG */
083  void rng_add_byte(unsigned char seed, unsigned entropy)
084  {
085     /* update the LFSR */
086     LFSR ^= seed;
087     LFSR  = (LFSR >> 8) ^ shiftab[LFSR & 0xFF];
088
089     /* credit the bits */
090     bit_count += entropy;
091
092     /* we use a .4 fixed point representation for entropy */
093     if (bit_count >= (32 << 4)) {
094        state[word_count++ & 7] ^= LFSR;
095        bit_count                = 0;
096     }
097  }
```

这个函数给RNG状态添加来自某些事件的一个字节的熵。首先，我们混入熵（第86行代码）并修改LFSR（第87行代码）。接着，我们记下熵（第90行代码）并检查是否可以把这个LFSR字加到状态中（第93行代码）。如果熵的数量等于或大于32位，我们就执行混合操作。

```
099  static void rng_churn_data(void)
100  {
101     unsigned char buf[64];
102     unsigned long x, y;
```

```
    /* update churn count and mix in */
    state[0] ^= churn_count++;

    /* store the state */
    for (x = 0; x < 8; x++) {
       STORE32L(state[x], buf + (x << 3));
    }

    /* copy the output pool as well (only 23 bytes) */
    for (x = 0; x < 23; x++) {
       buf[x+32] = pool[x];
    }
```

此时，局部数组buf包含有55个将要进行散列的字节数据。从前面的介绍可以知道，我们选择55个字节是为了让这个函数的效率能够尽可能的高。

```
    /* hash it */
    sha256_memory(buf, 55, pool);
```

这时我们调用SHA-256散列算法，它对*buf*数组中的55个字节数据进行散列，并且把32个字节的散列值存入*pool*数组。在这一步中并不需要考虑SHA-256是怎样工作的。

```
    /* mix the output directly into the state */
    for (x = 0; x < 8; x++) {
       LOAD32L(y, pool + (x << 3)); state[x] ^= y;
    }
```

注意到我们是将散列的输出和状态进行异或而不是替换状态。这可以防止回溯攻击。也就是说，如果我们只是简单地保留状态原来的值，那么一个能从输出得到状态的攻击者就可以向后或者向前执行这个PRNG。如果RNG用于阻塞模式，那么这就算不上是一个问题。

```

    /* reset states */
    pool_len   = 32;
    pool_idx   = 0;
    word_count = 0;
 }

 unsigned long rng_read(unsigned char *out,
                        unsigned long  len,
                                  int  block)
 {
    unsigned long x, y;

    x = 0;
    while (len) {
      /* can we read? */
      if (pool_len > 0) {
         /* copy upto pool_len bytes */
         for (y = 0; y < pool_len && y < len; y++) {
            *out++ = pool[pool_idx++];
         }
         pool_len -= y;
         len      -= y;
         x        += y;
```

```
      } else {
         /* can we churn? (or are non-blocking?)  */
         if (word_count >= 8 || !block) {
            rng_churn_data();
         } else {
            /* we can't so lets return */
            return x;
         }
      }
   }
   return x;
}
```

调用函数将使用读函数rng_read()从RNG系统中得到随机字节。根据*block*参数，它可以操作于两种模式中的任一种。如果*block*为非零值，那么这个函数就类似于一个RNG一样执行并且只读入缓冲池需要提供的字节数。不像一个真正的阻塞函数，它会返回部分读入字节而不是等待完全读满。如果它们需要传统的阻塞功能，那么调用函数不得不循环。这种灵活性经常是产生错误的一个原因，因为调用函数并不检查这个函数的返回值。不幸的是，即使返回给调用函数一个错误代码，调用函数也不会注意到，除非对程序的代码流程做一个较大的改变（例如，终止这个应用程序）。

如果*block*变量为0，那么这个函数就类似于一个PRNG一样执行，并且不管附加熵的数量是多少都会对已有的状态和缓冲池进行搅拌。如果状态内部含有足够的熵，就可以为任何目的把它作为一个PRNG运行一段合适的时间，就像运行一个合适的RNG一样。

警告 本章中出现的RNG并不是线程安全的，但至少兼容实时处理。尤其特别需要注意的是，如果把它直接加入到内核中，它可能会产生很大的破坏性。

函数rng_add_byte()和rng_read()都需要互斥锁以防止多个函数同时调用它们。解决这个问题的简单的办法是使用一个互斥锁装置。但是，要记住的是在实时处理平台上，如果互斥信号量是锁上的，那么你可能不得不丢弃rng_add_byte()函数调用以保持最低的延迟。

提示 如果你运行一个仅仅是基于rng_add_byte()机制的RNG事件俘获系统，状态可能已经含有超过256位的熵，而且你永远不会有一种办法来实现它。

针对这个问题的一种解决办法是在后台运行一个调用rng_read()的任务，并将数据存入一个调用函数可以访问的更大的缓冲区中。这可以让系统能够缓冲一个超过32个字节的有用数量的熵。大部分的加密任务最多只需要几百个字节的RNG数据。一个4KB大小的缓冲区用来平滑地移动数据绰绰有余。

3.4.4 RNG估算

至此，我们已经知道了收集什么，怎样处理它以及怎样产生输出。我们现在所需要的是捕获熵源并且保守估计它们的熵。理解那些可以有效地提取各种源的熵的模块是很重要的。对于所有的源，我们会给RNG提供中断（或者设备ID）以及一个高精度定时器的最低字节。在这些操作之后，我们将根据中断类型提供其他数据块列表。

我们首先从用户输入设备开始。

1. 键盘和鼠标

键盘是很容易捕获的。我们只是简单地需要键盘扫描码并把它们作为一个整体全部提供给RNG。对于大多数的平台，我们可以假设扫描码大都是16位长的。也可以针对你的平台进行相应的调整。在个人电脑（PC）的世界里，如果按下的键是典型的字母表中的一个字符，那么键盘控制器将发出一个字节。

当一个低频率的键被按下，例如功能按键、箭头按键或者小键盘上的按键，键盘控制器将会发送两个字节。无论如何，我们都可以假设扫描码的最低字节包含某种信息而上一个字节可能不包含。

在英语文本中，字符平均含有1.3位的熵，这个数据是通过统计重复的键和其他“常见”序列得到的。扫描码的低8位很有可能至少包含0.5位的熵。高8位可能为0，所以它的熵估计接近1/16位。类似地，中断码和高精度定时器源也分别含有1/16位的熵。

```
rng_src.c:
/* KEYBOARD */
void rng_keyboard(int INT, unsigned scancode, unsigned hrt)
{
   rng_add_byte(INT, 1);                         /* 1/16 bits  */
   rng_add_byte(scancode & 0xFF, 8);             /* 1/2 bits  */
   rng_add_byte((scancode >> 8) & 0xFF, 1);      /* 1/16 bits */
   rng_add_byte(hrt, 1);                         /* timer */
}
```

这段代码可以由键盘来调用，并且应该给它传递中断号（或设备ID）、扫描码以及高精度定时器的最低字节。显然，没有使用逻辑的PC AT扫描码应该进行合理的调整。一种不太周到的解决办法是，使用宿主语言取每个字符的平均熵并且至少把它除以2。

对于鼠标，我们基本上使用相同的规则，但与使用扫描码不同的是，我们使用鼠标位置和状态。

```
rng_src.c:
/* MOUSE */
void rng_mouse(int INT, int x, int y, int z,
               int buttons, unsigned hrt)
{
   rng_add_byte(INT, 1);                         /* 1/16 bits  */
   rng_add_byte(x & 255, 2);                     /* 1/8 bits */
   rng_add_byte(y & 255, 2);                     /* 1/8 bits */
   rng_add_byte(z & 255, 1);                     /* 1/16 bits */
   rng_add_byte(buttons & 255, 1);               /* 1/16 bits */
   rng_add_byte(hrt, 1);                         /* timer */
}
```

在这里我们加上鼠标的*x*、*y*以及*z*坐标（*z*坐标代表滚动轮）的低8位。我们估计*x*和*y*会给我们提供每位1/8th的熵，因为我们确实只对最低位感兴趣。例如，以一条竖线移动鼠标（假设你正移向文件菜单），你的*x*坐标不可能产生太大的变化。它可能会在向上的方向上稍微向左或向右移动一点，但它的大部分是不变的。当你试图把鼠标向上移动时，熵是通过鼠标是什么时候、在哪里以及*x*坐标的“误差”来产生的。

我们假设在这个函数中，鼠标按键（最高为8个）以*button*参数的低8位来压缩成布尔值。现实中，鼠标按键的状态包含很少的熵，因为大部分的鼠标事件都是用户为了点击而对某些东西进行定位的。

2. 定时器

定时器中断或者系统时钟都可以被捕捉以提取它们的熵。这里我们是寻找这两者之间的偏值。如果你的系统时钟和处理器时钟是互锁的（即它们是互为基础的），你应该跳过这个函数。

rng_src.c:
```
/* TIMER */
void rng_timer(int INT, unsigned timer, unsigned hrt)
{
   rng_add_byte(INT, 1);                      /* 1/16 bits */
   rng_add_byte(timer^hrt, 1);                /* 1/16 bits */
}
```

我们估计这两个定时器的异或（XOR）会产生1/16位的熵。

3. 通用设备

最后一个函数是针对通用设备的，例如存储设备或者网络设备，捕捉它们当中的用户数据所花费的代价是很高的。与前面不同的是，我们捕捉产生事件的设备ID以及当前高分辨率的时间。

rng_src.c:
```
/* DEVICE */
void rng_device(     int INT, unsigned minor,
                unsigned major, unsigned hrt)
{
   rng_add_byte(INT, 1);                      /* 1/16 bits */
   rng_add_byte(minor, 1);                    /* 1/16 bits */
   rng_add_byte(major, 1);                    /* 1/16 bits */
   rng_add_byte(hrt, 1);                      /* timer */
}
```

我们假设这些设备具有某种major:minor形式标识体系，例如Linux中的设备。如果major/minor无效的话，也可以使用USB或者PCI的设备ID，但是在这种情况下你不得不修改这个函数，从major和minor中都添加16位，而不是仅仅加上低8位。

3.4.5 RNG的设置

大部分平台所面临的一个重要的问题是，当它们第一次启动时是缺少熵的。在第一个用户空间程序运行时，不可能收集到很多的（如果有的话）事件。

Linux解决这个问题的办法是收集一个大小可变的RNG输出块并且将它写入一个文件中。在下一次启动时，文件中的位数据将以每位一位的速率添加到RNG中。需要注意的是，不要直接使用这些位作为RNG的输出并且在使用完之后尽快销毁它。

该方法存在安全风险，因为如果在种子文件被删除之前攻击者可以读取它的话，那么RNG中所有的熵都可能会暴露给攻击者。显然，必须以权限0400将这个文件标记为root（根）所有。

在存储一个种子也会成为一个问题的平台上，可能花费几秒的时间来读取一种类似于ADC之类的设备会更合适。以8kHz的5s的录制音频中，将至少含有256位的熵。解决这个问题的一

种简单的办法是收集这5s的音频，用SHA-256算法对收集到的数据进行散列，并以每位一位熵的速率将散列输出提供给RNG。

3.5 PRNG算法

我们现在已经有了构造一个RNG的好的开始。要记住的是大部分的平台，例如Windows、BSD系列和Linux各种发行版本都给用户提供了内核级别的RNG功能。如果有可能的话，使用那些而不要用自己所写的。

现在我们还需要快速的熵源。这里，我们把熵从一个通用的概念转化为一个专用的概念。从我们的角度来看，如果我们能够预测生成器的输出的话就很好，只要攻击者不能。如果攻击者不能预测那些位，那么对于他们来讲，我们就是有效地制造了随机位。

3.5.1 PRNG的设计

从高级别的角度来看，PRNG很像RNG。实际上，大多数流行的PRNG算法，例如Fortuna和NIST系列，都是可用作RNG的（当事件延迟不成为问题的时候）。

典型PRNG的处理流程图（如图3-3所示）和RNG的很相似，不同的是不需要收集这一步。任何输入的熵都是直接送给（更高的延迟）处理层并且它们马上就成为PRNG状态的一部分。

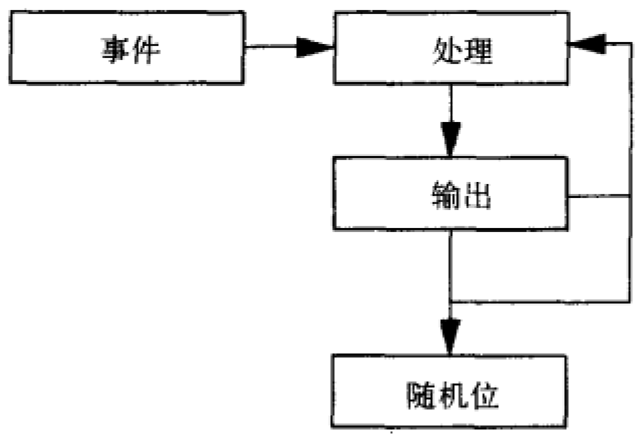


图3-3 PRNG处理流程图

大多数PRNG的目标都不同于RNG的。至少从外行的角度来看，高的熵输出需求仍然存在。我们所说的“外行”是指那些不知道PRNG算法内部状态的人。对于PRNG，他们用于迫切需要熵的系统，尤其是在那些没有足够多的用于输出层次处理能力的系统中。PRNG必须产生高的熵输出并且必须高效地完成。

1. 位提取器

形式化密码学称PRNG算法为“位提取器”或者“种子扩展器”。这是因为形式化模型（Oded Goldreich，Foundations of Cryptography，Basic Tools（中文译作《密码学基础》），牛津大学出版社，第一版）把它们看作是某种可以取种子并将其扩展到需要的长度的事物。实际上，你是将熵按照输出的长度进行传播。输出越长，一个辨别器就越有可能检测出来自一个带有特定种子算法的位数据。

2. 种子化（Seeding）及其生命周期

就像RNG，必须提供给PRNG种子以发挥所要求的功能。虽然大多数的PRNG都支持在初始化完成之后还可以进行重新种子化（reseeding），但并不是所有的都能这样，而且这也不是它们的安全威胁模型所要求的。一些PRNG，例如那些基于流密码的（例如RC4、SOBER-128和SEAL）并不直接支持重新种子化，且必须初始化以接受种子数据。

在大多数应用情况下，根据不同的应用，PRNG的用途可以分成两类。对许多短运行时应

用程序来说，例如文件加密工具，PRNG必须以短的时间周期生存并且还要对输出的长度不敏感。对长的运行时应用程序来说，例如服务器和用户守护程序，PRNG要有长的生命周期并且必须进行合理的维护。

对于短的一方面，我们将会研究Yarrow PRNG的一种派生算法，它可以很容易地通过一个分组算法和散列算法来构造。它也能像分组加密算法加密分组一样相对快速地产生输出。我们也将研究NIST的基于散列算法的DRBG函数，它更加复杂，但是对于使用了NIST加密的应用程序是必需的。

对于长的一方面，我们将研究Fortuna PRNG。它更加复杂并且很难设置，但它能够更好地适用于长时间的运行。特别是，我们将会看到Fortuna的设计是怎样抵抗状态发现攻击的。

3.5.2 PRNG的攻击

既然我们知道了一个看上去合理的PRNG是什么样的，但是我们能否把它拉开并攻破它？

首先，我们不得不理解攻破它是什么意思。PRNG的目的是产生位（或字节），以所有有意义的统计方法来看它们都是不可以预测的。如果攻击者能够经常预测出输出，那么这个PRNG就被攻破了。更精确地说，如果存在一种算法，它能够在可行的时间内从随机中辨别出PRNG，那么就是攻破。

PRNG中的攻破可以在几种场合下发生。我们能够把输入当作事件进行预测或者控制，以确保状态中的熵尽可能得低。另外一种办法是从输出回溯到内部状态，然后使用低熵的事件来欺骗用户从（未上锁的）PRNG中读入低熵的字节。

1. 输入控制

首先，我们来考虑控制输入。如果PRNG仅有的输入来自攻击者，那么所有的PRNG都会失败。熵的估算（正如Linux内核中所使用的）并不是一个有效的解决办法，因为一个攻击可能总是使用一个更高阶的模型来生成估算器会认为是随机的数据。例如，考虑一个仅看0^{th}阶统计的估算器，即值为1的位的数量和值为0的位的数量。一个攻击者能给PRNG提供一个01010101...数据流并且估算器对此一无所知。

增加模型的阶只是意味着攻击者不得不提前一步。如果我们使用1^{st}阶模型（计算位的对数），攻击者可以提供0001101100011011...数据流，如此，等等。实际上，如果攻击者控制了所有的熵输入，那么不管你选择了什么样的PRNG设计，都将会被成功地攻击。

这使得我们只要考虑攻击者可以控制某些输入的情况。基于下面的观察，能容易地证明这是可以成功的抵制这些情况的。假设你发送给PRNG 1位的熵（在1位中）并且攻击者可以给LFSR发送合适的数据，这样你的位就被取消出去了。根据熵的真正定义，这是不可能发生的，因为你的位具有不确定性。如果攻击者能够预测它，那么那个位的熵就不是1。实际上，如果你的输入当中真的含有熵，攻击者将不能把它们“取消”出去，即使LFSR确定是用来混合数据的。

2. 可塑性攻击（Malleability Attacks）

这些攻击类似于针对分组密码的选择明文攻击，不同的是这些攻击的目的是引导PRNG根

据选择的输入给出其内部状态。如果PRNG在处理层使用了任何数据依赖操作，攻击者就可以使用这种方法来控制程序的行为。例如，假设如果值为0和值为1的位之间没有均衡的话，你只对状态数据进行了散列。攻击者就能够利用这一点并提供可以避开散列的输入。

3. 回溯攻击

当输出数据泄露出PRNG内部状态的信息时，攻击者就能够向后对状态进行跟踪，这时回溯攻击就发生了。其目的是找出前一个输出。例如，如果一个PRNG是用来产生一个RSA密钥的，找出前一个输出就能够给攻击者提供这个RSA密钥的分解。

作为这种攻击的一个例子，假设PRNG只有一个LFSR。输出是其内部状态的一个线性组合。一个攻击者可以解决它，然后继续获取PRNG的前一个或者下一个输出。

即使PRNG算法的设计很好，即监听当前的状态并不能暴露前面的状态。例如，考虑rng.c中的RNG构造。如果我们删除第122行代码附近的XOR，那么当它被用作PRNG时，在每次调用之间状态并没有真正地变化。该意思是说，如果攻击者监听了状态并且我们没有在那里设置一个XOR，他就可以持续地向前运行并且也可以部分的向后回溯。

3.5.3 Yarrow PRNG

Yarrow设计是一个PRNG，它最初是用来作为一个长生命周期的广泛部署的系统。作为一个在各种类UNIX平台上的PRNG守护程序，它曾取得了一定程度的流行，但大概是因为Fortuna的出现，它已被归为一种快速的并且低劣的PRNG设计。

实际上，这个设计把熵和现有的缓冲池一同散列，然后这个缓冲池作为一个运行于CTR模式（参见第4章）的分组密码的对称密钥。该分组密码运行于CTR模式并且相当高效地产生PRNG输出。实际的Yarrow设计指出了怎样以及什么时候应该注意重新种子化的问题和怎样收集熵，等等。对我们的应用目的来讲，我们把它作为一个PRNG来使用，因此不需要关心种子是从哪里来的。也就是说，你应该能够设想这些种子具有足够的熵来解决你的威胁模型。

建议读者阅读它的设计论文，由John Kelsey、Bruce Schneier和Niels Ferguson所编写的“Yarrow-160: Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator”以获取精确的细节，因为我们这里的描述是相当简单的。

1. 设计

从图3-4和图3-5所示的分块流程图中，我们可以看到现有的缓冲池和种子一起进行散列来形成新的缓冲池（或者状态）。散列的使用避免了回溯攻击。假设种子数据的散列值只是简单地异或到缓冲池中，如果一个攻击者知道该缓冲池并且能猜到提供的种子数据，他就能够对状态进行回溯。

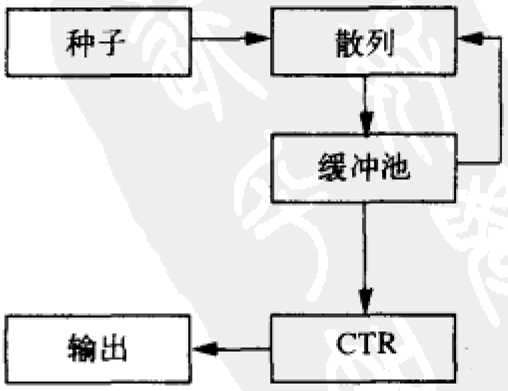


图3-4 简化的Yarrow PRNG分块流程图

这两块散列也可以用来防止现有的熵缓冲池因使用时间过长而造成的降级。也就是说，虽然缓冲池的散列并不增加熵，但它的确能修改CTR分组所使用的密钥。即使这些密钥可能是相关的（通过散列），也不可能找出它们之间的关系。CTR分组也不需要引入一个分组

密码，在CTR模式中，它有可能是一个散列算法。由于性能的原因，对于CTR模式最好使用一个分组加密算法。

在原始的Yarrow规范中，该设计需要SHA-1散列函数和Blowfish分组密码（或者仅有一个散列算法）。虽然这些并不是不好的选择，但在今天最好选择SHA-256和AES，因为它们更加现代、更加高效并且都是各种标准的一部分，包括FIPS系列标准。在这个算法中（如图3-6所示）我们把该缓冲池作为一个对称密钥，然后继续进行，以CTR模式对一个0字符串进行加密以产生输出。

输入：

pool：现有的缓冲池

seed：即将加到缓冲池中的种子

输出：

pool：新的修改过的缓冲池

1. *W* = *pool* || *seed*
2. *pool* = Hash(W)
3. return *pool*

图3-5 算法：重新种子化

输入：

pool：现有的缓冲池

IV：当前初始向量值

outlen：要读的字节数

输出：

output：随机字节

IV：新的初始向量值

1. *K* = 把*pool*作为一个分组密码的密钥进行调度（参见第4章）
2. D = CTR($\mathbf{0x00}^{outlen}$, *K*, *IV*)
3. Return *D*, *IV*.

图3-6 算法：Yarrow生成

符号$0x00^{outlen}$表示长度为outlen个字节的全0字符串。在第2步，我们调用了CTR函数，它还没有定义。这个函数的参数为<*plaintext, key, IV*>，其中*IV*初始为0并且在每个对重新种子化和生成函数的调用过程中一直保持不变。对同一个密钥重放*IV*是危险的，这也正是为什么对它保持修改是很重要的原因。随机字节在字符串*D*中，它是使用CTR函数对全0字符串加密的输出。

2. 重新种子化

原始的Yarrow规范要求和熵的源进行系统范围内的集成以支持Yarrow PRNG。这虽然不是一种不好的办法，但它并不是很适合这种Yarrow设计。我们将在对Fortuna的讨论中看到，有更好的从系统事件中收集熵的办法。

对于大多数短周期的密码学任务来说，使用一个系统RNG对Yarrow设置一次种子是安全的。一个实用的安全限制是使用一个单一的种子最多不超过1小时，或者不超过$2^{(w/4)}$个CTR分组，其中w是该分组密码的位长（例如，w=128的AES）。对于AES，这个限制为2^{32}个分组，或者是64GB的PRNG输出（在AMD Opteron处理器上，可能在远小于1个小时的时间内（大概500s）

生成2^{32}个输出。因此，这个限制确实具有现实的重要性并且应该牢牢记住）。

从技术上来讲，CTR模式的这个限制是生日悖论所规定的，按照其规定应该为$2^{(w/2)}$，之所以使用$2^{(w/4)}$是为了确保使用任何底层的分组密码都是可以的。目前，还没有比穷举更快的针对整个AES的攻击，但是那是会改变的，并且如果确实发生了，这将是一个重大的突破。把我们自己限制在一个很小的输出范围内将长久而有效地防止这个问题。

这些指导原则仅仅是建议。安全限制取决于你的威胁模型。在一些系统中，你可能会限制一个种子只使用1min或者只用于一个单一的输出。特别是在生成长期的证书以后，例如公钥认证，最好使当前的PRNG状态无效并立即进行重新种子化。

3. 保持状态

缓冲池和当前的CTR计数器值能够表示整个的Yarrow状态（参见第4章）。在一些平台中，例如嵌入式平台，我们也许希望保存这种状态，或者至少保留它所包含的熵以备将来之用。

取决于系统中是否有其他能够读取系统文件的用户，这可能是一个头疼的问题。正如大部分的Linux发行版本所使用的一样，典型的解决办法是从系统RNG中输出随机数据到一个文件中并且在启动时读取。这当然是一种针对这个问题的有效的解决办法。另外一种办法是对缓冲池进行散列并存储其散列值。对缓冲池自身进行散列可以直接捕获缓冲池中的所有熵。

从安全的角度来看，这两种技术都同等有效。剩下的威胁来自于一个已经读取了种子文件的攻击者。因此，在启动的任何可能的时候，引入一个新的种子是相当重要的。当本地用户不在或者无法读取种子文件的时候，种子文件的作用就显现出来了。这使得系统以PRNG状态的熵来启动，即使捕捉了很少或者一个都没有的事件。

4. 优点和缺点

Yarrow可以很高效地把一个种子转化为一个长的随机字符串。它非常依赖于散列算法的单向性和冲突约束，以及作为一个合理的伪随机置换的对称分组密码的行为。如果提供一个具有足够数量的熵的种子，完全可以使用Yarrow作为长期的运行。

Yarrow也很容易用来构造基本的密码学原子操作。这使得实现上更不可能出现错误，并且也降低了代码空间和内存的使用。

另一方面，Yarrow有一个很有限的状态，经常具有状态探索攻击的风险。虽然它们是不实际的，但的确存在这种理论上的风险。因此，应该避免长期或者在系统范围内使用它。

Yarrow可以很好地适用于许多短生命周期的任务，它们需要很小的熵。这些任务类似于命令行工具（例如GnuPG）、网络发送器以及小型服务器或者客户端（例如DSL路由盒）。

3.5.4 Fortuna PRNG

Fortuna是由Niels Ferguson和Bruce Schnier所提出的，它是对Yarrow设计的一个有效的升级。它仍然使用相当的CTR机制来产生输出，但与之不同的是它有更多的重新种子化元素以及一个更加复杂的缓冲池系统（参见Niels Ferguson和Bruce Schnier所著的*Practical Cryptography*，由Wiley出版社2003年出版）。

Fortuna拥有多个缓冲池并且只选择它们其中一个来生成CTR分组所使用的对称密钥，通过这种办法，Fortuna解决了Yarrow小的PRNG状态的问题。它更加适合于需要周期性的重新种子

化，并且能够抵抗可溯性和回溯攻击的长生命周期任务。

1. 设计

Fortuna设计通常是以你所要的用来收集熵的熵缓冲池的数量为特征的。这个数量取决于有多少事件以及你打算以什么频率来收集它们，应用程序将要运行多久以及你有多少内存。一个合理的缓冲池的数量是在4～32之间，后者是Fortuna设计的默认值。

当该算法开始之后（如图3-7所示），所有的缓冲池都会被有效地清零和清空。缓冲池计数器*pool_idx*表示当前我们正指向哪个缓冲池；*pool0_cnt*表示添加到第0个缓冲池中的字节数；*reset_cnt*表示PRNG已经被重新种子化（重置CTR密钥）的次数。

这些缓冲池实际上并不是数据的缓冲区，在实际应用中，它们被实现为用于接收数据的散列状态。我们使用散列状态来替代缓冲区，这使得添加大量的数据并不需要浪费内存。

当熵被添加到缓冲池中以后（如图3-8和图3-9所示），它是以两个字节的头部开始，这个头部包含一个ID字节和一个长度字节。ID字节是一个开发者选择的用以区分不同熵源的简单标识符。长度字节表示被添加的熵的字节数。熵被逻辑地添加到第*pool_idx*个缓冲池，我们将看到这等于加上缓冲池的散列值。如果加上第0个缓冲池，我们需要把*pool0_cnt*加上已添加的字节的个数。接着，我们让*pool_idx*递增并且在需要时重置为0。如果在把熵加到第0个缓冲池的过程中它超过了一个最小的大小（即64个字节），就应该调用重新初始化算法。

输入：

Numpools: 所用的熵缓冲池的数量

输出：

pool: 缓冲池数组

pool_idx: 缓冲池索引

pool0_cnt: 缓冲池0中的字节数

reset_cnt: 重新种子化的数目

IV: 当前的密码IV

K: 对称密钥

1. For *j* from 0 to NUMPOOLS − 1 do
    1. *pool*[*j*] = null
    2. *pool_idx* = 0
3. *pool0_cnt* = 0
4. *reset_cnt* = 0
5. *IV* = 0
6. *K* = null
7. return <*pool, pool_idx, pool0_cnt, reset_cnt, IV, K*>

图3-7 算法：Fortuna 初始化

输入：

pool: 现有的缓冲池

seed: 要添加的熵

seedID: 应用程序（或者事件）指定的种子标识符

pool0_cnt: 第0个缓冲池中的字节数

pool_idx: 当前缓冲池索引

输出：

pool: 修改过的缓冲池

pool0_cnt: 第0个缓冲池中当前的字节数

pool_idx: 新的缓冲池索引

1. *W* = *seedID* || length(*seed*) || *seed*
2. Add *W* to the *pool*[*pool_idx*]
3. If *pool_idx*= 0 then *pool0_cnt* = *pool0_cnt* + length(*seed*)
4. If *pool0_cnt* >= 64 call reseed algorithm.
5. *pool_idx* = (*pool_idx* + 1) mod *NUMPOOLS*
6. return <*pool, pool0_cnt, pool_idx*>

图3-8 算法：Fortuna添加熵

注意，在图中我们使用了一个含有4个缓冲池的系统。Fortuna绝不会有这种限制。我们选择4个是为了使图表小一些。

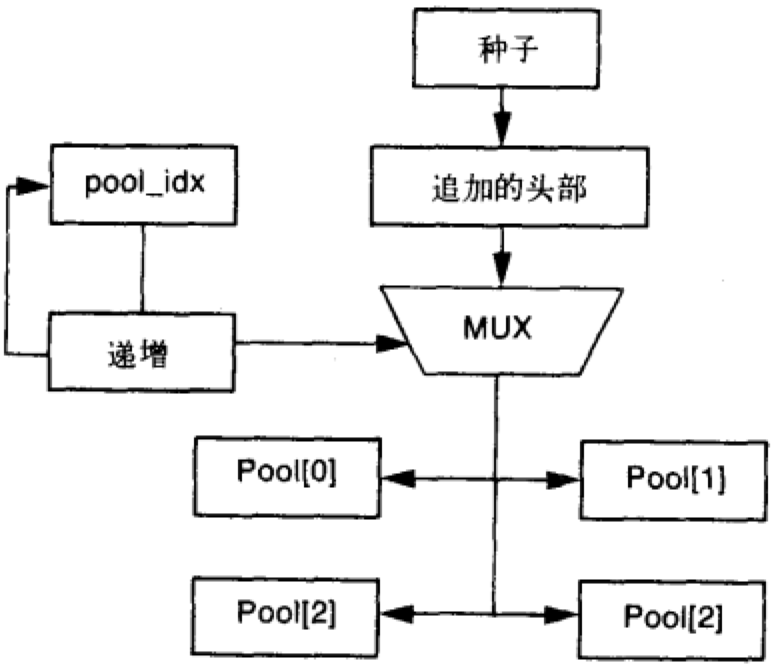


图3-9 Fortuna熵添加流程图

重新种子化将从选中的缓冲池中取出熵并将其转化成一个分组密码将要使用的对称密钥以产生输出（如图3-10和图3-11所示）。首先，*reset_cnt*计数器进行递增。*reset_cnt*的值可以解释为一个位掩码，也就是说，如果位x为1，那么在这个算法中将使用缓冲池x。所有被选中的缓冲池都将被散列，所有的散列值都将和已有的对称密钥进行连接（按照从第一个到最后一个的顺序），并且把散列字符串的散列值作为对称密钥。所有被选中的缓冲池都将被清空并且设为0。第0个缓冲池总是被选中。

输入：

pool: 现有的缓冲池

K: 当前的对称密钥

reset_cnt: 当前的重置计数器

输出：

pool: 修改过的缓冲池

K: 新的对称密钥

reset_cnt: 修改过的重置计数器

1. *reset_cnt* = *reset_cnt* + 1
2. *W* = *K*
3. for *j* from 0 to NUMPOOLS − 1
   1. if (*j* = 0) OR (((1<<*j*) AND *reset_cnt*) > 0) then
      - i. *W* = *W* || hash(*pool*[*j*])
      - ii. *pool*[*j*] = null
4. *K* = Hash(*W*)
5. return <*pool*, *K*, *reset_cnt*>

图3-10 算法：Fortuna重新种子化

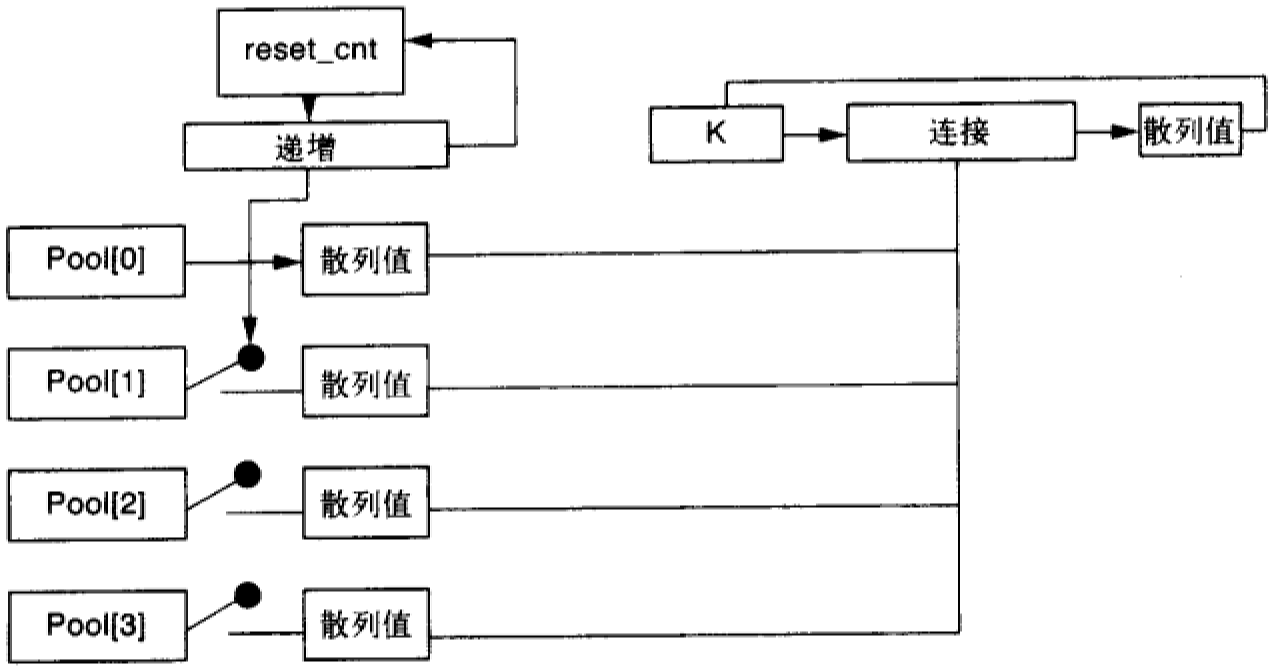


图3-11 算法：Fortuna重新种子化流程图

从Fortuna中提取熵是相当简单的。首先，如果一个给定的时间已经过去或者已经做了一些读操作，那么重新种子化函数就会被调用。典型的，作为一个系统范围内的PRNG，其时间延迟应该很短，例如，每隔10s。如果没有定时器或者这不是作为一个守护程序来运行的，你就可以对这个函数进行10次调用。

当重新种子化处理完成后，CTR模式的分组密码能够生成调用函数所要求的足够多的字节。当算法开始时，分组密码的IV初始为0，然后在算法的整个生存周期中运行。计数器以little-endian形式处理，分别从字节0到字节15进行递增。

当生成所有的输出之后，分组密码将同步两次以为了下一次的使用而对自身重新设置密钥。在Fortuna的设计中，使用了一种AES的候选分组密码，但我们可以直接使用AES（Fortuna是在AES算法最终确定之前进行设计的）。两次同步产生256位，这是最大的密钥大小允许值。然后新的密钥会被调度并且用于后续的读操作。

2. 重新种子化

当正在添加熵时，Fortuna是不会执行重新种子化操作的，而是以循环的形式把数据简单地和其中一个缓冲池进行连接（如图3-5所示）。Fortuna打算从许多源中收集熵，就像我们在rng.c中描述的系统RNG一样。

在那些中断延迟并不是一个很重要的问题的系统中，Fortuan可以很容易地作为一个系统RNG使用。在有限的熵添加应用程序中（例如命令行工具），Fortuna降为Yarrow并且减去了重新设置密钥的位。正是因为这一个原因，Fortuna不适合用于短生命周期的应用程序中。

一个Fortuna能够作为合理选择的经典例子是在一个HTTP服务器中。它得到许多的事件（请求），并且这些请求可以用可变的（不确定的）时间来完成。用这些信息片，以及偶尔使用系统RNG和其他系统资源，来对Fortuna进行种子化能够产生一个相对有效并且非常安全的用户空间PRNG。

3. 保持状态

Fortuna PRNG的整个状态可以用缓冲池的当前状态、当前的对称密钥和各种计数器来描述。

对在关闭的时候要保存什么来说，它比Yarrow要复杂得多。

简单地释放出少数字节并不会用到还没有被使用的下一个缓冲池中的熵。最简单的办法是使用*reset_cnt*来进行一次重新种子化，*reset_cnt*等于所有被1减的1型，并且所有的缓冲池都将受到对称密钥的影响。然后，从PRNG中释放出32个字节到种子文件中。

如果你有空间，那么把每个缓冲池的散列值存储到种子文件中有助于在下次加载时保持PRNG的寿命。

4. 优点和缺点

对于长生命周期的系统来说，Fortuna是一种好的设计。它通过进行长的位抽取来散发熵，针对状态探索攻击提供前向安全。它是基于普通受到肯定的设计概念，这使得对它的分析是很简单的。

Fortuna最适合于服务器和后台形式的应用程序，在这些程序中它可以收集到许多的熵事件。反之，它不适合于短生命周期的应用程序。它的设计比Yarrow更加复杂，而且对于短生命周期的应用程序来说这是完全没有必要的。这种设计在数据和代码方面使用了更多的内存空间。

3.5.5 NIST的基于散列的DRBG

NIST的基于散列的DRBG（Deterministic Random Bit Generator，确定性的随机位生成器）是3个在NIST SP 800-90中最新提出的随机函数的其中之一。在这里我们只讨论在规范的第10.1节中描述的Hash_DRBG算法。这个PRNG有一个参数集，它定义了算法中的各种变量（如表3-1所示）。

表3-1 Hash_DRBG中的参数

	SHA-1	SHA-224	SHA-256	SHA-384	SHA-512
支持的安全强度	80	112	128	192	256
highest_supported_security_strength	80	112	128	192	256
输出分组的长度 **(outlen)**	160	224	256	384	512
实例化和重新种子化所需要的最小熵 最小的熵输入长度 **(min_length)**	*security_strength*				
最大的熵输入长度 **(max_length)**	$< 2^{35}$ bits				
Hash_DRBG的种子长度（seedlen）	440	440	440	888	888
max_personalization_string_length	$< 2^{35}$				
max_additional_input_length	$< 2^{35}$				
max_number_of_bits_per_request	$< 2^{19}$				
reseed_interval	$< 2^{18}$				

1. 设计

Hash_DRBG的内部状态由以下部分组成：

- 一个*seedlen*位长的值*V*，它在每次对DRBG的调用中都会被修改；

- 一个*seedlen*位长的常数*C*，它取决于*seed*；
- 一个计数器*reseed_counter*，它表示在实例化或者重新种子化的过程中从得到新的*entropy_input*开始，伪随机位的请求次数。

该PRNG通过Hash_DRBG实例化过程进行初始化（见规范中的10.1.1.2节）。

这个算法返回一个有效的状态，Hash_DRBG函数的其他部分都可以用它来工作（如图3-12所示）。它依赖于函数hash_df()（如图3-13）所示，这个函数我们已经做了定义。在此之前，我们先来看重新种子化函数。

输入：

entropy_input: 从一个熵源中得到的位串

nonce: 另一个熵源（或者来自一个PRNG不重复的源）

输出：

工作状态<*V*,*C*,*reseed_counter*>

1. seed_material = entropy_input || nonce
2. *seed* = Hash_df(*seed_material*, *seedlen*)
3. *V* = *seed*
4. *C* = Hash_df((**0x00** || *V*), *seedlen*)
5. reseed_counter = 1
6. Return *V*, *C*, *reseed_counter*

图3-12 算法：Hash_DRBG实例化

输入：

<*V*, *C*, *reseed_counter*>: 当前工作状态

entropy_input: 将要添加到状态中的新位串

additional_input: 标识应用程序的位串（可以为null）

输出：

新的工作状态 <*V*, *C*, *reseed_counter*>

1. seed_material = 0x01 || V || entropy_input || additional_input
2. *seed* = Hash_df(*seed_material*, *seedlen*)
3. *V* = *seed*
4. *C* = Hash_df((0x00 || *V*), *seedlen*)
5. *reseed_counter* = 1
6. Return <*V*, *C*, *reseed_counter*>

图3-13 算法：Hash_DRBG 重新种子化

PDG

这个算法（如图3-14所示）使用新的熵并且将它混合入DRBG的状态中。该算法接受以应用程序指定位形式的附加输入。它可以为null（空），而且一般来说最好是这样。它对已有的缓冲池（V）和新的熵一起进行散列，这一点和Yarrow很相似。为了在技术上遵循规范，种子必须为seedlen位的长度。这一点就足以让这个算法用起来很特别。

输入：

<*V, C, reseed_counter*>：当前工作状态

requested_number_of_bits：将要返回的位数

additional_input：应用程序指定的输入

输出：

status：表示调用是否成功的状态

returned_bits：返回的位数

新的工作状态： <*V, C, reseed_counter*>

1. 如果*reseed_counter* > *reseed_internal*，那么就返回一个需要重新种子化的标志
2. 如果(*additional_input* 不为 null)，那么
    1. w = Hash(**0x02** || *V* || *additional_input*)
    2. $V = (V + w) \bmod 2^{seedlen}$
3. *returned_bits* = Hashgen(*requested_number_of_bits, V*)
4. *H* = Hash(**0x03** || V)
5. $V = (V + H + C + reseed_counter) \bmod 2^{seedlen}$
6. *reseed_counter* = *reseed_counter* + 1
7. 返回 success, *returned_bits* 和新的 <*V, C, reseed_counter*> 值

图3-14 算法：Hash_DRBG生成

这个函数使用工作状态并且提取出一个位串。它使用一个用Hash()表示的散列函数和一个我们还没有提到的新函数Hashgen()。非常类似于Fortuna，在返回（第5步）之前这个算法会修改缓冲池（V）。这是用来防止回溯攻击的。

这个函数执行了一个把输入串延伸为输出所需要的位数（如图3-15所示）。这和算法Hashgen（如图3-16所示）相似而且现在为什么当这两个函数执行类似的功能时都需要存在还不是很明显的。

这个函数为了生成函数对输入的种子进行延伸。实际上它和Hash_df（如图3-15所示）非常相似以至于可以和另一个相混淆。显著的区别是，计数器是加到种子上的而不是和它进行连接的。

2. 重新种子化

Hash_DRBG的重新种子化应该遵循和Yarrow类似的规则。因为仅有一个缓冲池，所有来自于源的熵都会立即使用。这使得长期使用它是不可取的，例如，Fortuna。

PDG

输入：

input_string: 将要被散列的串

no_of_bits_to_return: 将要返回的位数

输出：

status: 表示调用是否成功的状态

requested_bits: 返回的位数

1. 如果 no_of_bits_to_return > max_number_of_bits，那么就返回一个错误
2. *temp* = null 串
3. *len* = *no_of_bits_to_return* / *outlen* 循环
4. *counter* = **0x01**
5. for *j* = 1 to *len* do

   temp = *temp* || Hash(*counter* || *no_of_bits_to_return* || *input_string*)

   counter = *counter* + 1
6. *requested_bits* = leftmost(*no_of_bits_to_return*) of *temp*
7. 返回 success 和 *requested_bits*

图3-15 算法：Hash_df

输入：

requested_no_of_bits: 将要返回的位数

V: V的当前值

输出：

returned_bits: 返回的位数

1. *m* = *requested_no_of_bits* / *outlen* 循环
2. *data* = *V*
3. *W* = null
4. for *j* = 1 to *m* do
   1. *w*[j] = Hash(*data*)
   2. *W* = *W* || *w*[j]
   3. *data* = (*data* + 1) mod 2^{outlen}

   5. *returned_bits* = leftmost(*requested_no_of_bits*) of *W*
   6. return *returned_bits*

图3-16 算法：Hashgen

3. 保持状态

这个PRNG的状态由*V*, *C*, 和*reseed_counter*变量所组成。在这种情况下，最好只用生成函

数来生成一个随机串并且把它保存在你的seedfile中。

4. 优点和缺点

Hash_DRBG函数的确比Yarrow更加复杂而且没有Fortuna通用。它处理一些Yarrow不能处理的问题，如状态探索攻击。但是，它也在输入长度方面要求相当严格（例如种子），并且也有一些很明显地无用的重复操作。我们应该注意到NIST SP 800-90在写这本书的时候，仍然是一个草案而且有可能会有所修改。

从积极的方面看，该算法比Yarrow更加健壮并且包含一些不错的调谐和优化，这使得描述和实现它都很简单。这个算法和SP 800-90系列是值得关注的。不幸的是，在写这本书的时候，SP 800-90的注释期已经结束而且还没有出现新的草案。

3.6 总结

3.6.1 RNG与PRNG

在设计一个加密系统时首先要解决的是针对生成随机位的问题，你需要指出是需要一个RNG还仅仅是一个可以很好地进行种子化的PRNG。如果你在制造一个完完全全自我包含的产品，你将会很明确的需要一个RNG。如果你的应用程序是在一个平台上运行的，例如Windows或者Linux，那么用来对一个应用程序PRNG进行种子化的最好选择是一个系统RNG。

典型的，一个RNG在两种环境下是有用的：缺少永久性存储器以及需要紧密的信息理论属性。在没有合适的永久性存储器的环境下，你不能从一台设备（或者应用程序）的运行时发送一个种子到另一个。你可以使用那些被称为熔丝位（fuse bits）的来为PRNG得到一个初始状态，但是如果重用它会导致一种不安全的处理。在这些场合中，需要一个RNG，至少要在每次运行时对PRNG进行初始种子设置。

在其他环境中，一个应用程序需要移除这样一种假设，即PRNG产生和真正的随机位没有区别的位。比如一个认证签名机构确实应该使用一个RNG（或者一些）作为随机位的来源。假设在某些时候PRNG是很好地种子化了的任务是相当重要的。

熔丝位

熔丝位实际上是ROM的一种形式，它是在tape-out（tape-out是一个集成电路设计的最后一个阶段的名字，在这个时候，电路的详细说明将会发送给制造商）的掩模阶段（masking stage）之后生成的。电路的每个实例会有一个惟一的并且完全熔入它们自身的随机位类型。这些位将作为PRNG的初始状态并且允许执行跟踪和诊断操作。PRNG的执行能够判断其正确性，而且一台相应的宿主设备能够生成相同的位（如果需要这样的话）。

熔丝位是不可修改的。这意味着你的设备必须不能断电，或者必须要有一种形式的永久性存储器以存储修改过的PRNG状态的拷贝。也有一些可以解除这种限制的办法。例如，可以将熔丝位和一个实时的时钟输出一起散列来得到一个新的PRNG工作状态。如果这个实时的时钟输出是不可信任的（例如，能够被回溯到前一个时间），那么这可能会不安全，如果只有它一个的话，那么在别的方面却是安全的。

3.6.2 PRNG的使用

在对RNG的要求不是很严格的环境中，PRNG也许是一种更加合适的选择。PRNG很明显地要比RNG更快、有更小的延迟并且在大多数情况下能够提供和RNG同样有效的安全性。

应用程序中PRNG的生命周期以至少一次的重新种子化操作开始。在大多数的平台上，都有一个系统范围的RNG。一个很短的至少对256个位数据的读操作就能够提供足够的种子材料来启动一个在不可预测的（从敌人的角度来看）状态中的PRNG。

尽管你至少需要一次重新种子化操作，但是不建议在应用程序的生命周期中进行重新种子化。在实际应用中，甚至具有很少或者没有熵的种子都可以安全地提供给任何安全的PRNG重新种子化函数。理想的情况是，在任何一个具有很长的生命周期的事件结束之后强制进行一次重新种子化，例如，用户证书的生成。

当带有加密服务的应用程序执行完以后，最好保存状态并且/或者从内存中清除状态。在具有RNG的平台上，它并不总是有助于把输出写到种子文件中。当应用程序在RNG能够产生输出之前（例如在启动时）就需要熵的话，就会发生例外。在这种情况下，种子文件就不应该一字不差地包含PRNG的状态；也就是说，不要直接把状态写到种子文件中。相反，写到种子文件中的应该是调用PRNG的生成函数来产生256个位（或者更多）的结果，或者是当前状态的一个单向散列值。这意味着即使一个攻击者能够读取已保存的状态，但仍然不能恢复出前一个输出。

来自安全研究人员的忠告

种子化要做的和不可做的

要做的

- 至少重新种子化一次，并且要使用由一个可信任的RNG所产生的至少256位的数据；
- 如果可能的话应经常重新种子化；
- 在生成长周期的证书之后要进行重新种子化；
- 要对状态进行散列或者针对种子文件生成位数据；
- 在使用完状态后要清除其数据。

不可做的

- 不要用常数来重新种子化；
- 不要用少于256位的RNG熵来重新种子化；
- 不要总是相信用户的输入（例如捕捉点键事件）；
- 不要对延长的时间周期在没有重新种子化的情况下使用相同的PRNG状态；
- 不要在任何时候泄漏PRNG的状态；
- 不要为了以后的使用而把PRNG状态作为一个种子文件来使用。

3.6.3 示例平台

1. 桌面和服务器

桌面和服务器平台对基本组件的使用通常是相似的。服务器通常通过它们更高的规范和它们是典型的多种方式（例如多处理器）这样一种事实来区分。它们共享如下这些组件。

- 高分辨率的处理器定时器（例如，RDTSC）；
- PIT、ACPI和HPET定时器；
- 声音编解码器（例如，Intel HDA或者AC'97兼容系列）；
- USB、PS/2、串行和并行端口；
- SATA、SCSI和IDE端口（带有独立的存储设备）；
- 网络接口控制器。

大多数服务器上的负载保证了一个稳定的存储以及网络中断数据流可以被捕捉。典型的，桌面会有足够的存储中断来使得一个RNG能够运行。

在中断很少的环境中，定时器偏差和ADC噪声选项是经常存在的。这和一个用来捕捉1s的音频信号并将其提供给RNG的启动脚本一样普通。定时器中断的提供可以保证至少有一个逐渐增长的熵数据流。

我们提到了在任何机器上都可以访问的各种端口，例如USB、PS/2、串行和并行端口，因为有不同的制造商出售可以提供宿主随机位的RNG设备。最流行的设备是SG100（www.protego.se/sg100_en.htm）9针串行口RNG。它使用由一个二极管按照某种阈值产生的噪声。它们也有一个针对现代系统的USB变体（www.protego.se/sg200_d.htm）。也有其他的一些设备，例如来自FDK的RPG100B IC（www.fdk.co.jp/cryber-e/pi_ic_rpg100.htm），这是一个小的32位插针芯片，它能够产生多达250kbit/s的随机数据。

这些平台通常需要种子文件，因为在启动刚刚结束之后，一般并没有足够的熵来对PRNG进行种子化。在系统关闭程序中，种子文件必须被修改或者删除以避免重用相同的种子。一种安全的处理方法是在启动的时候应删除这个种子文件，用这种方法，如果机器在运行过程中突然崩溃了，也是没有办法来重用种子的。

2. 控制台

大多数的视频游戏控制台在考虑安全相关问题之前都是被很好地设计以及生产了的。Sony PS2带来了不带RNG的宽带适配器，意思是说大多数的连接都可以是畅通无阻的。类似地，Sony PSP和Nintendo DS不带任何加密设置就能够实现网络的兼容。

在这些情况下，你可以很容易地说“只要使用一个RNG就可以了”，除非这时你没有一个可以直接获得的RNG。你所拥有的是一个相当好的中断和干扰源，以及用来存储种子数据的永久性存储器空间。这些平台上的熵都是以用户输入的形式出现的，通常是来自一台某种类别的带有定时器数据的控制器。先前提到的RNG可以足够的快以使得这些操作对性能的影响降低到最小。在Nintendo DS中有一个麦克风，它也能够从ADC噪声中提供相当数量的熵。

因为性能的原因，让RNG在游戏的整个生命周期中一直运行并不是理想的做法。用户可以忍受在RNG初始化的时候有些慢，但不会忍受在玩游戏的时候也慢。在这种情况下，最好是使用RNG来给PRNG进行种子化并且对游戏中所需的任何熵都用它。通常，只是在建立连接的时候会用到熵，在这之后加密系统就会运行并且不再需要熵。

在这些平台上的种子管理会有些棘手。理想情况下，你想把熵保存下来以备下一次的运行使用，而用户却喜欢玩起来的速度很快并且要保证交易是安全的。到2006年6月，所有流行的控制台都具有存储数据的功能，或者使用一个内部的硬盘，一个外部的闪存，甚至是在游戏磁

带匣（game cartridge）内部（例如在Nintendo DS磁带匣里）。在这些平台上处理这种问题的技巧是，当你有了足够的熵并且在用户没有随机的关掉电源之前就保存一个新的种子。在大多数平台上，电源按钮是软件驱动的并且能够发出一个中断信号。但是，我们也不得不处理电源断电或者其他的关闭，例如崩溃。

在这些情况下，最好是在启动之后使种子无效，以避免再次使用它。当程序运行时，RNG会收集数据，并且一旦熵足够时就应该对PRNG进行初始化。这时PRNG的生成函数就能够用来生成一个新的种子文件。

特别地，写到本地的数据对于“伪装的眼睛”是安全的。也就是说，你可以把种子记下来，然后在将来的运行中把它作为你的核心熵源来使用。但是，这个种子的熵可以降级，并且这个进程并不是用来租用的（例如Nintendo DS游戏）。在PRNG种子化之前，最好是在RNG中至少收集256位额外的熵并且写入新的种子文件中。

3. 网络设备

网络设备面临着和控制台一样的“战斗”。它们通常没有大量的存储器，并且在第一次使用时其内部状态还没有准备好。一个典型的网络设备可能是一些像DSL路由器，或者访问设备（像RFID锁）一样的东西。在某些客户端或者服务器端以及它们自身之间，需要密码学算法来进行认证并对数据进行加密。例如，你可能使用SSL来登录你的DSL路由器，你的远程照相机可能会使用SFTP来把文件传到你的桌面上，并且你的RFID门必须使用一个中央访问控制服务器来对一个试图进行的访问进行认证。

这些应用相对于软件的RNG来讲，更加适合硬件RNG的情况，因为它们是典型的自定义设计。设计一个二极管RNG到电路中比设计一个需要一个单向函数的完全有限状态机更加简单。更糟的是，大多数的网络设备都没有足够的中断来生成任何可以度量的熵。

3.7 常见问题

下面的常见问题，由本书的作者所回答，它们即可以用来测试你对本章所出现的概念的理解，也可以帮助你在现实生活中实现这些概念。如果希望作者解答你的问题，请浏览www.syngress.com/solutions，然后点击“Ask the Author”表单。

问：什么是熵？

答：熵就是在一个可以观察的事件中对不可确定性的一种度量，并且特别地以位为单位来表示，因为经常以二进制决策图来描述它。例如，一个完美的硬币投掷就是所谓的有1位的熵，因为它的结果无论是正面朝上或反面朝上都是以同样的概率出现的。

问：什么叫一个事件？

答：一个事件就RNG或者PRNG算法可以观察的某些事情，例如硬件中断。和事件相关的数据，例如哪个事件以及它是什么时候发生的，可以作为它们的熵来提取。

问：我在哪里能够找到一个标准的RNG设计？

答：很抱歉，还没有由公共机构托管的（公开的）RNG设计。他们所做的标准只是RNG测试。你的设计必须通过它们，才能成为FIPS 140-2所认证的。实际上，即使这样还不够。为了通过认证，就必须在每次它启动的时候都要自我测试。这是为了确保硬件不出问题。

问：这看起来很像魔术？

答：是的，RNG设计实质上是科学以及推测。在你的鼠标移动中有多少位的熵呢？这很难确切地说出来，这也是为什么要建议进行适当地估算的原因。

问：为什么我不能一直只使用一个RNG？

答：RNG是比PRNG算法更慢并且经常阻塞（而PRNG算法很少阻塞）。这意味着在一个应用程序的运行时环境中，很难使用一个RNG。对于大部分的应用目标来说，PRNG能够保证马上返回一个结果，这是更好的，也是一个事实。

问：在短的和长的生命周期的PRNG之间有什么实际重要性？

答：Yarrow，至少是在本文中提到的，是可以安全地用于短的和长的生命周期的，因为已经证明它可以经常性的进行种子化。Fortuna在一个更长的运行时中传播它的输入熵，这使得它更加适合于长期运行的程序。但是，在实际当中，已证实如果RPNG被很好地种子化并且你不需要生成大量的位，其输出是可以安全使用的。

问：有没有其他的PRNG标准？

答：NIST SP 800-90指定了一个基于分组密码、HMAC和椭圆曲线的PRNG。ANSI X9.31指定了一个三重DES位生成器，它正在进行修改以支持AES。

问：有没有什么是真正随机的？

答：有的，量子力学中允许处理的标准模型是真随机的。关于这方面，最常被人们引用的例子是元素的放射性衰变。

通过在计算机中加入一个Geiger计数器钩子就可以从放射性衰变中提取出大量的熵。熵是通过度量衰变对之间的时间长度来提取的。如果第一对衰变之间的时间比第二对的短，那么就记录一个0。如果第一对衰变之间的时间比第二对的长，那么就记录一个1。用这种方式来生成的随机位具有非常接近于每位1位的熵。

现在还不可能存在一个针对放射性衰变的类型。即使有的话，正如我们所知道的，对于量子力学来说现在这还是不可能的。针对这个问题，也可以考虑另外一个事实，那就是量子力学已经被证实只有百万分之一的影响因素。如果量子力学的其中一个基础理论是错误的，那么这将会非常令人震惊。

除了量子力学之外，在确定的系统中也非常有可能包含熵。考虑一个质点以精确的每秒5的平方根米的速度进行运行。我们不能计算出5的平方根的所有十进制数的位置，因为它是一个无理数。这意味着对于任何有限的展开式，在它的值中总是有一个很小的度的熵（当然，我们也可以多做一些运算以得到下一个十进制数的位置，因此这种展开式就不适合于密码学应用）。

第4章

高级加密标准

本章解决方案：

- 什么是高级加密标准
- 分组密码
- AES的设计
- 实现
- 实用的攻击
- 链接模式
- 总结

☑ 总结
☑ 快速查找解决方案
☑ 常见问题

4.1 简介

高级加密标准（Advanced Encryption Standard，AES）开始于1997年的一项公告，这个公告是由NIST发出的，它寻找一个用来替代年代久远而且不再安全的数据加密标准（Data Encryption Standard，DES）算法的。这个时候，DES已经多次被证明是不安全的，并且为了在新的标准中激发信心，他们征求于公众再一次提交新的设计。DES使用56位的秘密密钥，这意味着穷举攻击是可能的并且也是实用的。为了有一个更大的密钥空间，AES不得不是一个128位的分组密码；也就是说，一次处理128位的输入明文分组。AES也必须支持128、192和256位的密钥并且要比DES更加高效。

今天看来，对分组密码的这些强制性限制看起来相当冗繁。我们采用AES是为了其能够适用于几乎每种密码学应用场合。但是，在20世纪90年代，大部分的分组密码例如IDEA和Blowfish仍然是64位的分组密码。虽然它们支持比DES更大的密钥，NIST仍然继续寻找在将来更加高效并且实用的设计。

这个号召的结果是，人们一共提交了15个设计，其中仅有5个（MARS、Twofish、RC6、Serpent和Rijndael）进入了下一轮的筛选。其他的10个因为效率和安全性的原因而被拒绝。在2000年年末，NIST宣布Rijndael将成为最终的AES算法。这个决定是基于在第三次选票中Rijndael获得了大部分的选票（差额选票是公平的），而且它也受到了NSA的认可。从技术上来说，NSA宣称所有的5个候选算法作为AES都是足够安全的，并不仅仅是Rijndael。

Rijndael是由两个比利时密码学家Joan Daemen 和 Vincent Rijmen所设计的。它实际上重用了Square分组密码算法并且对它进行了重新设计以抵抗已知的攻击。在AES的筛选过程中它特

别吸引人，这主要是因为它的效率（它是最通用的高效设计之一）和很好的密码学特性。Rijndael是一种遵循Daemens博士的宽迹（wide-trail）设计理论的替代-置换网络（Substitution-Permutation Network，SPN）。已经证明它即可以抵抗线性也可以抵抗差分密码分析（攻破DES的攻击方法），并且它在其他方面也有相当好的统计特性。实际上，Rijndael是5个最终候选算法中惟一的一个能够符合这些要求的算法。另外一个也安全的，Serpent，推测也能抵抗同样的攻击，但是它没有Rijndael那样受到欢迎，因为它实再是太慢了。

Rijndael（或者现在叫AES）是专利免费的，并且其发明者也给出了各种公开的参考实现。他们的快速实现实际上是大多数软件AES实现的基础，包括OpenSSL、GnuPG和LibTomCrypt。本书中的快速及半快速的实现是基于Rijndael的参考代码。

4.1.1 分组密码

在继续研究AES设计之前，我们应该讨论一下什么是分组密码（block cipher），以及它们在密码学中充当什么样的角色。

术语分组密码实际上是用来和通常的流密码（stream cipher）进行区分的。一个密码算法是一个隐藏消息的含义的过程。起初，它们是以简单的替代密码的形式，然后是流密码的形式，它们可以对消息的单独符号进行编码。在实际应用方面，它经常使用轮转机以及后来出现的移位寄存器（类似于LFSR）。这种理论试图找到一个具有很好的平衡性质的长周期生成器，而且通过一个非线性函数对输出进行过滤以创建一个密钥流。不幸地是，许多近来相关的研究使得大多数的基于LFSR的密码都不再安全。

分组密码和流密码的区别在于它是在一步当中对一组符号进行编码。从明文到密文的映射对于一个给定的秘密密钥来说是固定的。也就是说，使用同样的秘密密钥，相同的明文将映射成相同的密文。大多数常用的分组密码大都使用64位或者128位的分组大小。意思是说它们以64位或者128位的分组来处理明文。对于更长的消息，可以通过多次调用密码来进行编码，通常是使用一种链接模式（chaining mode），例如CTR以保证消息的保密性。由于映射大小的缘故，分组密码可以实现成一个很大的查找表（如图4-1所示）。

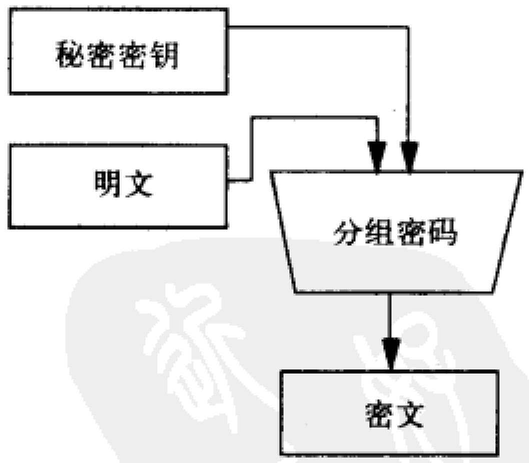


图4-1 分组密码示意图

早期的分组密码包括IBM设计小组所设计的（DES和Lucifer）以及在20世纪80年代和90年代早期的那些过多的设计。在AES于1997年开始之后，这种提交给大会的设计方案彻底消亡了。早期的分组密码系列是对64位的分组进行编码的而且密钥也很短，仅在64位左右。一些设计例如IDEA和Blowfish打破了这种模型并使用了更大的密钥。大多数密码的基本设计都是比较一致的：找到一种非线性函数并且用它对明文进行多次的迭代，这使得在没有密钥的情况下从密文映射回明文是很困难的。作为对比，DES有16轮同样的函数，IDEA有8轮，RC5起初有12轮，Blowfish有16轮，AES有10轮（现在一致认为RC5仅在16轮或者更多轮的时候才是安全的。虽然通常应该默认使用AES，但在代码空间可能是一个问题的地方也可以使用RC5）。通过使用一个算法来执行这种映射，分组密码可以非常

简洁、高效并且几乎在任何地方都可以使用。

从技术上来讲，分组密码就是密码学家所谓的伪随机置换（Pseudo Random Permutation，PRP）。也就是说，如果你能让每种可能的输入都通过这个密码算法，那么你将会得到实际上是输入的一种随机置换的输出（这是分组密码作为一个双射的结果）。秘密密钥控制了置换的顺序，而且不同的密钥应该选择看起来是随机的置换。

不严格地说，从安全性的角度来看，一个“好的”分组密码是一个在知道了其置换（或者部分置换）的情况下，除了穷举搜索之外不能得到密钥的算法；即一个攻击者，虽然他收集了关于置换顺序的信息，但是他不可能找到一个比尝试所有可能的密钥更快的找出密钥的方法。目前，人们相信AES所支持的3种密钥大小都具有这种性质。

尽管一个分组密码的行为非常类似于一个随机置换，但是它不应该用在自己身上。因为对于一个给定的密钥，其映射是静态的，即同样的明文分组将映射成相同的密文分组。这意味着，一个用于已加密的数据的分组密码在某些环境下可以直接泄漏出相当多的数据。

幸运地是，已经证实，如果我们假设分组密码是一个相当好的PRP，那么我们就可以用它来构建各种事物。首先，我们可以创建链接模式，例如CBC和CTR（稍后讨论），这可以允许我们获得所需要的保密性但又不暴露出明文的某些自然特征。我们也可以构造混合加密以及消息认证码，例如CCM和GCM（参见第7章）来获得保密性以及同步认证。最后，我们也可以用来构造PRNG，例如Yarrow和Fortuna。

分组密码的用途是相当广泛的，这使得它们对于各种问题都有相当的吸引力。我们将在第5章中看到散列算法也同样的用途广泛，并且也将知道什么时候在这两者之间根据手中掌握的问题进行取舍。

4.1.2 AES的设计

AES（Rijndael）分组密码（参见http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf）接受一个128位的明文，并且在一个128、192或者256位秘密密钥的控制下产生一个128位的密文。它是一个替代－置换网络的设计，并且带有一个称作轮（round）的步骤的集合，其中轮数可以为9、11或者13（取决于密钥长度），这样可以将明文映射为密文。

一轮AES由下面的4步组成：

1. SubBytes
2. ShiftRows
3. MixColumns
4. AddRoundKey

每一轮分别使用它自己的128位轮密钥（round key），它是由秘密密钥通过一个称为密钥调度（key schedule）的过程处理而产生的。不要低估一个设计合理的密钥调度方案的重要性。它把密钥的熵散发给每一个轮密钥。如果熵没有被很好地传播，就会产生各种麻烦，例如等价密钥、相关密钥以及其他类似的分别征服攻击（distinguishing attack）。

AES把128位的输入看作是一个由16个字节组成的向量，并用一个4 × 4的列矩阵（big-endian）的形式来组织，叫做状态（state）。即第一个字节映射为$a_{0,0}$，第3个字节映射为$a_{3,0}$，第

4个字节为$a_{0,1}$，第16个字节映射为$a_{3,3}$（如图4-2所示）。

整个AES分组密码由下面的步骤组成。

1. AddRoundKey（round=0）
2. for round=1到Nr−1（9、11或者13，这取决于密钥的大小） do
   - SubBytes
   - ShiftRows
   - MixColumns
   - AddRoundKey（round）
3. SubBytes
4. ShiftRows
5. AddRoundKey（Nr）

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$

图4-2 AES的状态示意图

1. 有限域（Finite Field）数学

AES实际上能够定义为在域GF（2）的元素上的一系列标量和向量的运算。这并不需要完全严格地掌握伽罗瓦域（Galois Field）的知识来理解这些运算。但是，大致了解该算法的实现是很有帮助的。在你第一次阅读本小节时，如果不是太理解的话，不要着急。对该算法的大部分来说，除非你真地想对AES进行优化（尤其是针对硬件实现），否则你并不需要理解这一节来有效地实现AES。

GF（p）意思是一个特征为p的有限域。这是什么意思呢？首先，我们对代数群做一个快速的浏览。我们将在对GCM和公钥算法的介绍中更加详细地讨论它。一个群是一些元素（可以是数字、多项式、椭圆曲线上的点，或者其他任何你可以用一个群运算以阶来描述的事物）的集合，并且加上一个明确定义的群运算，还要有一个幺元、零元和逆元。

如果一个群的群运算是加法，那么这个群就叫做环；需要注意的是，这种加法并不是严格意义上的整数加法。一个环的例子是用Z来表示的整数环。这个群包含了所有的负数和正数。我们可以用一些整数模一个整数来构造一个有限环，一个经典的例子是时钟算术——即整数模12。在这个群中，共有12个元素，从整数0到整数11。也有一个幺元，0，它同时也是零元。每个元素a都有一个逆元——a。我们将在椭圆曲线密码学中看到，曲线上两个点的加法也可以是一个群运算。

如果一个环也有乘法运算，那么这个环就是一个域。一个群的域元素传统上叫做单元(units)。域的一个例子是有理数（a/b形式的数）。可以构造一个有限域，例如，对一些整数模一个素数。例如，在整数模5的域中，有5个元素（0～4）以及4个单位元（1～4）。每个元素遵循有限环的规则。另外，每个单位元还遵循有限域的规则。同时域中还存在一个乘法逆元，叫做元素1。每个单位元a都有一个乘法逆元，定义为$1/a$模5。例如，2模5的乘法逆元是3，因为2×3模5=1。

简单来说，这意味着我们有一群元素，可以进行加、减、乘、除运算。特征为p的意思是一个域，有p个元素（p−1个单位元，因为0不是单位元）模素数p。GF(p)仅当p为素数时才有意义，但是它也可以扩展到扩展域，表示为$GF(p^k)$。扩展域典型地用于某些基本多项式形式的实现，也就是说，$GF(p^k)$中的一个元素就是一个用GF(p)中的元素作为系数，度为$k-1$的多项式。

AES使用域GF(2)，它实质上是整数模2所形成的域。但是，它扩展到了多项式域中，通常用$GF(2^8)$来表示，而且有时也把它看成是一个向量，用$GF(2)^8$来表示。作为最小的单位元，AES处理8位的数量，这经常记作$GF(2^8)$。从技术上来说，这是不正确的，因为它应该写成GF(2)[x]或者$GF(2)^8$。这种表达式表示一个含有8个GF(2)元素的向量，或者简单地说是8位。

对于初次接触用GF(2)[x]来表示的多项式初学者来说，可能会有些麻烦。它是一个以GF(2)中的元素为系数的多项式集合。这个表达式的意思是说，我们把含有8个位的向量看成是一个度为7的多项式的系数（例如，$<a_0、a_1、a_2、a_3、a_4、a_5、a_6、a_7>$可以转化为多项式$p(x) = a_0x^0 + a_1x^1 + \cdots + a_7x^7$）。

为了构造一个域，我们需要使用一个能够不被任何一个具有相同的底（GF(2)）的度数更小的多项式整除的多项式。特别地，这类多项式称为不可约多项式（irreducible polynomials）。我们把整个域定义为GF(2)[x]/v(x)，其中v(x)是不可约多项式。在AES当中，它选择了多项式$v(x)=x^8 + x^4 + x^3 + x + 1$。当v($x$)的各个位以整数存储时，它表示值0x11B。那些想要了解更多的读者也许听说过本原多项式（primitive polynomial），意思是说多项式g(x)=x能够生成域中所有的单位元。我们说生成，意思是存在某个值k，使得$g(x)^k=y(x)$对于所有的GF(2)[x]/v(x)中的y(x)都成立。在AES当中，多项式v(x)并不是本原的，但是多项式g(x)=x+1是这个域的生成多项式。

这个域当中的加法就是一个简单的异或（XOR）运算。如果两个输入的度数都为7或者更小，那么就不需要约简。否则，结果必须通过模v(x)来约简。乘法就是乘以其他的多项式。如果我们想找到一个c(x)=a(x)b(x)，只需要简单地找到一个向量$<c_0, c_1, c_2, \ldots, c_{15}>$，其中$c_n$等于乘积中相当度数的系数之和。在C语言中，乘法可以通过下面的程序来实现。

```
/* return ab mod v(x) */
unsigned gf_mul(unsigned a, unsigned b)
{
   unsigned res;
   res = 0;
   while (a) {
    /* if bit of a is set add b */
    if (a & 1) res ^= b;

    /* multiply b by x */
    b <<= 1;

    /* reduce it modulo 0x11B which is the AES poly */
    if (b & 0x100) b ^= 0x11B;

    /* get next bit of a */
    a >>= 1;
  }
  return res;
}
```

这个算法是一个普通的double-and-add算法（类似于我们将在RSA中看到的square-and-multiply算法），它用来计算AES中所用的域中的ab的积。在该算法的循环中，我们首先测试a的最低位是否为1。如果是，我们将b的值加到res上。接着，使用一个移位运算来计算x乘以b的值。如果一个位向量代表这个多项式，那么在右边插入一个0位，其值也是不变的。然后，我们通过模AES的多项式来对b进行约简。实质上，如果第8位为1，我们就从该值中减去（或者

异或）模数。

这个多项式域也会用在SubBytes和MixColumn步骤中。在实际应用中，至少是在软件实现中，我们并不用这种办法来实现乘法，因为这样做会很慢。

2. AddRoundKey

轮函数的这一步是把轮密钥加到状态中（在GF(2)中）。它执行了16个并行的把密钥加到状态中的运算。加法是通过异或运算来完成的（如图4-3所示）。

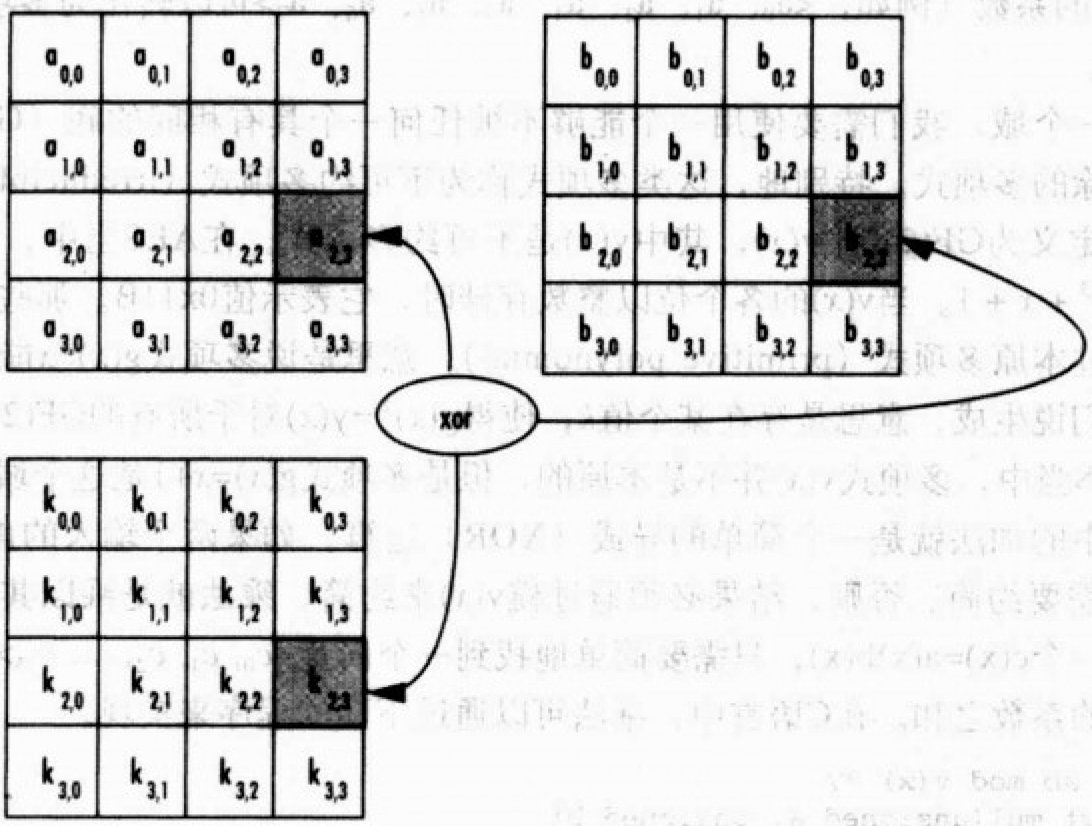


图4-3 AES AddRoundKey函数

其中的*k*矩阵是一个轮密钥并且对每一轮都有一个惟一的密钥。因为密钥的加法是一个简单的异或，所以它常实现为在32位软件中从列开始的一个32位的异或操作。

3. SubBytes

轮函数的SubBytes步骤是用来执行SPN中的非线性混淆步骤的。它把16个字节的每一个都并行地映射为一个新的字节，这是通过一个两步骤的替代操作来完成的（如图4-4所示）。

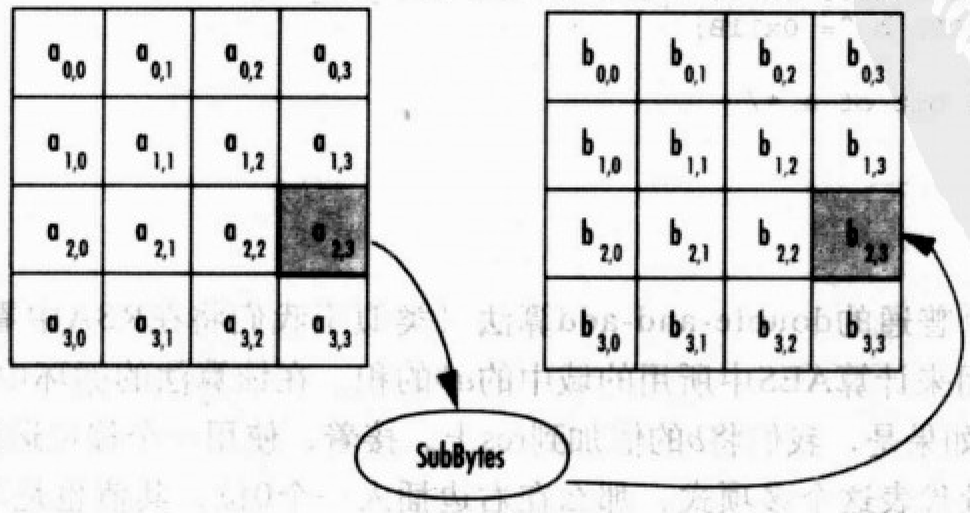


图4-4 AES SubBytes函数

替代是由一个在GF(2)[x]/v(x)上的乘法逆操作和一个在$GF(2)^8$上的仿射变换（如图4-5所示）所组成。一个单位元a的乘法逆元是另一个单位元b，使得ab模AES的多项式和多项式p(x)=1是同余的（相等，或者当用$v(x)$约简时相等）。对于AES，我们定义了一个特例，即a(x)=0的逆是它自身。

有几种方法可以找出逆元。因为该域是很小的，所以需要平均128次乘法运算的穷举就可以找到它。用这种办法，我们简单地用域中所有的单位元去乘以a，直到积为0。

```
unsigned gf_inv_brute(unsigned x)
{
   unsigned y;
   if (x == 0) return 0;
   for (y = 1; y < 256; y++) {
      if (gf_mul(x, y) == 1) return y;
   }
}
```

$$
\begin{bmatrix}
1&0&0&0&1&1&1&1\\
1&1&0&0&0&1&1&1\\
1&1&1&0&0&0&1&1\\
1&1&1&1&0&0&0&1\\
1&1&1&1&1&0&0&0\\
0&1&1&1&1&1&0&0\\
0&0&1&1&1&1&1&0\\
0&0&0&1&1&1&1&1
\end{bmatrix}
\begin{bmatrix}
x_0\\x_1\\x_2\\x_3\\x_4\\x_5\\x_6\\x_7
\end{bmatrix}
+
\begin{bmatrix}
1\\1\\0\\0\\0\\1\\1\\0
\end{bmatrix}
$$

图4-5 AES仿射变换

它也可以由幂规则来找到。域GF(2^8)的阶是$2^8-1=255$且$a(x)^{254}=a(x)^{-1}$。$a(x)^{254}$的计算可以通过8次平方和7次乘法来完成。我们把它们单独列出来，因为GF(2)中的平方运算是一个O(n)时间的操作（反乘法所需要的O(n^2)相反）。

```
unsigned gf_inv_power(unsigned x)
{
   unsigned y, z;
   y = 1;
   for (z = 0; z < 7; z++) {
      y = gf_mul(gf_mul(y, y), x);
   }
   return gf_mul(y, y);
}
```

这里我们使用gf_mul来执行平方操作。但是，还有更快的而且更加适合于硬件的方法来完成这个任务。在GF(2)[x]中，平方运算是一个简单的移位操作，并且在输入位之间插入0位。例如，值1101_2变为10100010_2。在平方之后，还需要进行一次约简。在软件实现中，至少对于AES来说，用于完成这个函数所需要的代码空间大到和SubBytes本身差不多。在硬件中，平方运算是通过另一种实现技巧来完成的，我们将在稍后进行讨论。

也可以用欧几里德算法来找到它，最后通过使用一个对数和反对数表。对于软件实现来说，这些都很不合适。最好的办法是把SubBytes和ShiftRows以及MixColumns混合起来（稍后将进行讨论），或者完全用一个单独的8×8的查找表来实现。

在逆操作完成之后，这8个位被发送到仿射变换中去处理并且其输出就是SubBytes函数的结果。仿射变换用来表示，其中向量$<x_0, x_1, x_2, \cdots, x_7>$表示从最低位到最高位的8个位。联合起来，针对8位值的SubBytes替代表如图4-6所示。

SubBytes的逆是仿射变换的逆，然后是乘法逆元。它可以很容易地用一个短的循环从SubBytes中推导出。

```
int x;
for (x = 0; x < 256; x++) InvSubBytes[SubBytes[x]] = x;
```

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1x	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2x	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3x	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4x	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5x	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6x	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7x	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8x	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9x	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
Ax	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
Bx	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
Cx	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
Dx	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
Ex	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
Fx	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

图4-6 AES SubBytes表

4. 适用于硬件环境下的SubBytes

这一小节讨论了一个针对硬件实现的技巧，它对于软件没有什么用处（感谢Elliptic Semiconductor——www.ellipticsemi.com提供的信息）。我们之所以在这里讨论它，一方面是因为这个信息传播并不广泛，另一方面如果你是硬件的行业人士，这对你也许很有用。

适用于硬件环境下的SubBytes实现利用了这样一个事实的优点，即你可以通过计算一个很小的逆，然后对输出应用某些普通的GF（2）数学运算来计算乘法逆元。这使得电路可以变得更小，并且在大多数情况比一个8×8的ROM表实现更快。

这个算法的一般流程如下所示。

(1) 对8位的输入应用一个前向线性映射

(2) 把8位的结果分成两个4位的字*b*和*c*（*b*代表最高位往下开始的4位）。

(3) 计算$d=((b^2 \times r(x)) \times OR(c \times b) \times OR\ c^2)^{-1}$。

(4) $p=b \times d$。

(5) $q= (c \times OR\ b) \times d$。

或者你也可以使用$q=(c \times d) \times OR\ p$。

(6) 对8位字p||q应用逆线性映射。

其中的乘法是$GF(2)[x]/t(x)$上的乘法并且要模$t(x)=x^4 + x^1 + 1$，$r(x)$的值为$x^3 + x^2 + x$，且其模逆（第三步）是一个4位的模$t(x)$的逆。前向线性映射如下面的8×8矩阵定义（在GF(2)上的）。

1	0	0	0	0	0	0	0
0	1	1	0	0	1	0	0
0	1	0	1	0	0	1	0
0	0	0	0	0	0	1	0
1	0	0	1	1	1	0	0
1	0	0	0	1	0	1	1
1	0	0	0	1	1	0	0
0	0	1	0	0	1	1	1

其逆线性映射是在GF(2)上的逆矩阵变换。逆映射如下所示。

1	0	0	0	0	0	0	0
0	0	1	1	1	0	1	0
0	0	0	0	0	1	1	1
0	0	0	0	1	0	1	0
1	1	1	1	1	1	1	1
0	1	1	1	1	1	0	1
0	0	0	1	0	0	0	0
0	1	1	0	1	0	1	1

这个技巧也利用了另外一个事实的优点，即平方实质上是一个自由运算。如果进行平方的输入是向量$<x_0, x_1, x_2, x_3>$（x_0是最低位），那么下面的这4个等式可以指定输出向量。

$$y_0 = x_0 \text{ XOR } x_2$$
$$y_1 = x_2$$
$$y_2 = x_1 \text{ XOR } x_3$$
$$y_3 = x_3$$

输出向量$<y_0, y_1, y_2, y_3>$是输入向量的平方再模上多项式t(x)。你也可以以并行的方式实现这个4位的乘法，因为它们是很小的。乘以r(x)的运算是固定的，因此它不需要实现为一个完整的GF(2)[x]上的乘法。4位的逆操作即可以用一个ROM查找表来实现，即分解（递归使用这种技巧），或者让合成器来对它进行优化。

在对这8位的输入使用了这个6步的算法后，我们现在可以求出模逆了。然后应用AES仿射变换来完成SubBytes变换。

5. ShiftRows

ShiftRows这一步对状态中的每一行分别进行向左循环移动0、1、2和3个位置。它是完全线性的（如图4-7所示）。

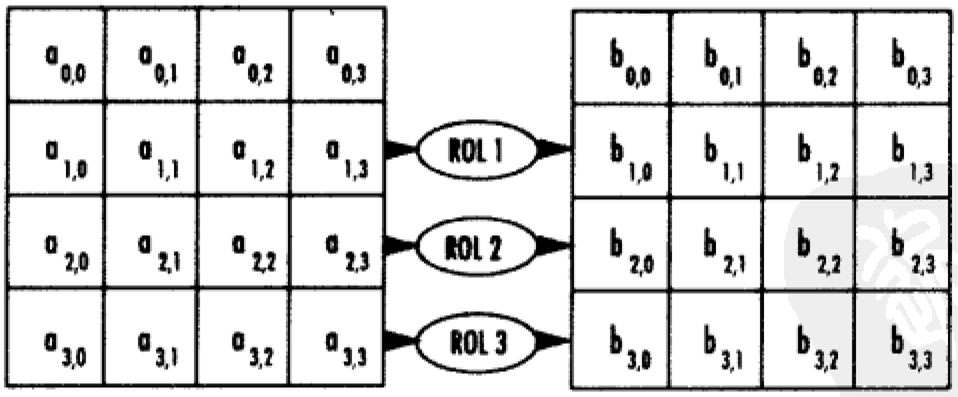


图4-7 AES ShiftRows函数

在实际应用中，我们将会看到这是通过重命名（renaming）来实现的而不是一个实际的移动。也就是说，通过对字节移动的替代，我们只是简单地在得到它们的地方对它们进行修改就可以。在32位的软件中，我们可以很容易地把ShiftRows和SubBytes以及MixColumns相混合，而不用来回交换字节。

6. MixColumns

MixColumns这一步对状态中的每一列乘上一个4 × 4的变换，这个变换就是所谓的极大距

离可分码（Maximally Distance Separable，MDS）。这一步的目的是扩大差别并且让输出线性依赖于其他输入。即如果一个单一的输入字节在两个明文中发生了改变（输入中所有其他的字节没变），这个改变将尽可能快地传播到状态中的其他字节中去（如图4-8所示）。

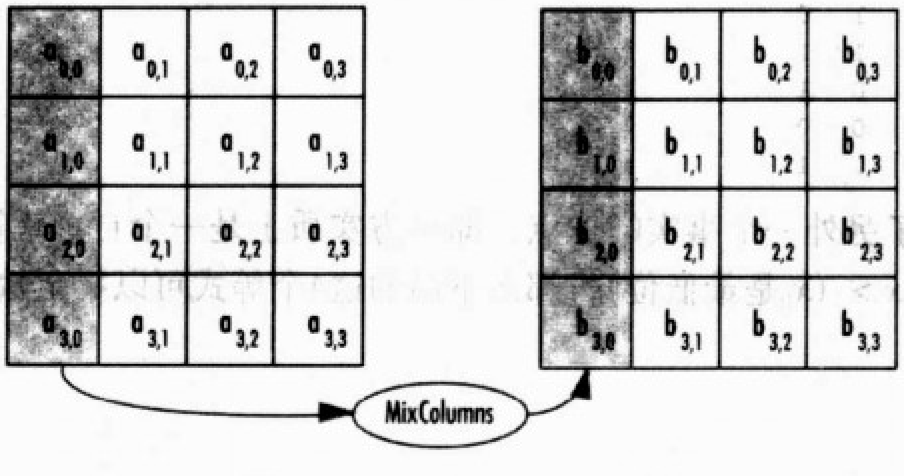


图4-8 AES MixColumns函数

在AES中，选择了一个MDS矩阵来完成这项任务。MDS变换实际上形成了编码理论中的某些事物，例如错误校验码（Reed-Solomon）。它们具有一个独特的性质，在两个不同的*k*空间输入之间，不同的输入和输出坐标的和总是至少为*k*+1。例如，如果我们在两个4字节的输入之间抛出一个输入字节中的1位，那么我们可以预测输出的差别至少为（4+1）−1个字节。

MDS码尤其吸引分组密码的构造，因为它们经常是很高效的而且具有很好的密码学性质。目前，一些流行的分组密码使用了MDS变换，例如Square、Rijndael、Twofish、Anubis和Kazhad（后面的两个是NESSIE标准工程中的一部分）。

SubBytes函数给该密码算法提供了非线性的组件，但是仅仅是并行地对状态中的字节进行操作，它们互不影响。如果AES把MixColumns删除，这个算法就能够很容易地攻破，并且这个算法就变得毫无用处。

MDS矩阵是由向量<2, 3, 1, 1>生成的，它在图4-9中进行了说明。

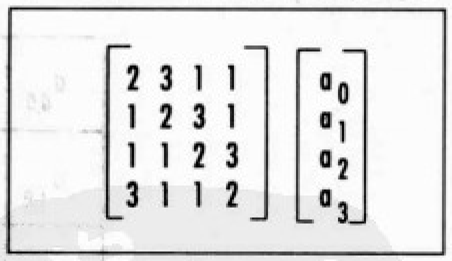


图4-9 AES MDS矩阵

其中的乘法是在GF(2)[*x*]/v(*x*)上执行的，同样地，SubBytes也是在它上面执行的。实际值1、2和3直接映射成多项式。例如，2=*x*，3=*x*+1。

根据目标平台的不同，MixColumns可以由各种不同的方法来实现。最简单的办法是使用一个函数，这个函数执行一个乘*x*的操作，通常叫做*xtime*。

```
unsigned xtime(unsigned x)
{
   /* multiply by x */
   x <<= 1;
   /* reduce */
   if (x & 0x100) x ^= 0x11B;
   return x;
}

void MixColumn(unsigned char *col)
{
```

```
    unsigned char tmp[4], xt[4];
    xt[0]    = xtime(col[0]);
    xt[1]    = xtime(col[1]);
    xt[2]    = xtime(col[2]);
    xt[3]    = xtime(col[3]);
    tmp[0]   = xt[0]  ^ xt[1]  ^ col[1] ^ col[2] ^ col[3];
    tmp[1]   = col[0] ^ xt[1]  ^ xt[2]  ^ col[2] ^ col[3];
    tmp[2]   = col[0] ^ col[1] ^ xt[2]  ^ xt[3]  ^ col[3];
    tmp[3]   = xt[0]  ^ col[0] ^ col[1] ^ col[2] ^ xt[3];
    col[0]   = tmp[0];
    col[1]   = tmp[1];
    col[2]   = tmp[2];
    col[3]   = tmp[3];
}
```

这个函数以4个字节的偏移来访问数据，这对应于AES状态矩阵的宽。这个函数将被调用4次，每次以一个字节的col偏移。在实际应用中，状态将会被放到另一个缓冲区中，以避免双缓冲区需求（例如，从tmp[ ]复制到col[ ]）。在典型的硬件实现中，最直接的办法往往是选择的办法，因为它可以用并行的方式实现而且只有很短的一个关键路径。

也有另外一种办法在软件中实现MixColumn，这实际上使得我们把SubBytes、ShiftRows和MixColumn整合在一个单独的操作集中进行。首先，考虑没有ShiftRows或者SubBytes的轮函数。

MixColumn函数是一个32 × 32的线性函数，这可以用4个8 × 32的查找表来实现。我们创建4个8 × 32的表，如果其他3个输入字节是0的话，那么其中输出的每个字节（对所有的256个字而言）表示MixColumn的积。下面的代码可以为我们生成使用MixColumns()函数的表。

```
unsigned char mc_tab[4][256][4];
void gen_tab(void)
{
    unsigned char col[16];
    int x, y, z;

    for (y = 0; y < 4; y++) {
      for (x = 0; x < 256; x++) {
        for (z = 0; z < 16; z++) col[z] = 0;
        col[y] = x;
        MixColumn(col);
        for (z = 0; z < 4; z++) {
            mc_tab[y][x][z] = col[z];
        }
      }
    }
}
```

现在，如果我们通过以littleendian格式来加载每4个字节的向量，将mc_tab映射到一个4 × 256的32位字的数组中的话，那么我们可以用如下的代码来计算MixColumns。

```
unsigned long MixColumn32(unsigned long col)
{
    return mc_tab[0][col&255]        ^
           mc_tab[1][(col>>8)&255]   ^
           mc_tab[2][(col>>16)&255]  ^
           mc_tab[3][(col>>24)&255];
}
```

需要注意地是，我们现在是假设mc_tab是一个32位字的数组。这样可以工作，因为矩阵代数是可交换的。假设这个MixColumns变换的矩阵是C，而且我们有一个输入向量<a, b, c, d>，那么这个优化就说我们可以计算出C<a, b, c, d>的积为C<a, 0, 0, 0>+C<0, b, 0, 0>+…+C<0, 0, 0, d>。实际上，对于所有的256个a的值来说，$mc_tab[0][a]$等于$[2, 1, 1, 3]\times a$。因为向量的每一个组件只是一个8位的变量，我们可以简单地使用一个表来查询。我们对a所有的可能值进行预计算C<a, 0, 0, 0>（称之为$mc_tab[0]$）。我们然后对C<0, b, 0, 0>、C<0, 0, c, 0>和C<0, 0, 0, d>做同样的事情。用这种办法，每个表需要1KB的存储空间，4个表一共需要4KB。

对于这种办法还可以有一些折衷。由于矩阵的自然属性，mc_tab[1]中的字实际上等于mc_tab[0]中的字向右循环移动8位。类似地，mc_tab[2]中的字是循环移动16位，mc_tab[3]循环移动24位。这使得我们通过以时间来换空间，把表空间压缩到1KB（这也能帮助抵抗某些形式的时间攻击，稍后将进行讨论）。

$$\begin{bmatrix} 14 & 11 & 13 & 9 \\ 9 & 14 & 11 & 13 \\ 13 & 9 & 14 & 11 \\ 11 & 13 & 9 & 14 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

图4-10 AES InvMixColumns矩阵

MixColumns的逆是图4-10中所示的变换。

```
void InvMixColumn(unsigned char *col)
{
   unsigned char tmp[4];
   tmp[0]   = gf_mul(col[0], 14) ^ gf_mul(col[1], 11) ^
              gf_mul(col[2], 13) ^ gf_mul(col[3], 9);
   tmp[1]   = gf_mul(col[1], 14) ^ gf_mul(col[2], 11) ^
              gf_mul(col[3], 13) ^ gf_mul(col[0], 9);
   tmp[2]   = gf_mul(col[2], 14) ^ gf_mul(col[3], 11) ^
              gf_mul(col[0], 13) ^ gf_mul(col[1], 9);
   tmp[3]   = gf_mul(col[3], 14) ^ gf_mul(col[0], 11) ^
              gf_mul(col[1], 13) ^ gf_mul(col[2], 9);
   col[0]   = tmp[0];
   col[1]   = tmp[1];
   col[2]   = tmp[2];
   col[3]   = tmp[3];
}
```

正如我们所看到的，前向变换具有更为简单的系数（只有很少的位为1），它会在当选择在域中怎样使用AES的时候就会产生作用。其逆变换同样也可以用表来实现，同样的压缩技巧也可以用于这个实现。总的来说，只需要2KB的表来实现这种压缩的方法。在8位平台上，对gf_mul()的调用可以用表查找来代替。总之，只需要1KB的内存。

7. 最后一轮

AES的最后一轮（第10、12或者14轮，这取决于密钥的大小）和其他的轮是不同的，不同之处在于它使用如下的步骤：

(1) SubBytes

(2) ShiftRows

(3) AddRoundKey

8. 逆算法

其逆算法实质上是由相当顺序的步骤所组成，只不过我们把每一步都替换成了它们的逆。

(1) AddRoundKey(Nr)

(2) for round = Nr-1 downto 1 do

1) InvShiftRow
2) InvSubBytes
3) AddRoundKey(round)
4) InvMixColumns

(3) InvSubBytes
(4) InvShiftRow
(5) AddRoundKey(0)

在这些步骤中，“Inv”前缀的意思是这是逆操作。根据实现的不同，密钥调度也稍微有些不同。我们将会看到在快速的AES代码中，把AddRoundKey移到轮的最后一步可以使得我们创建一个类似于加密的解密程序。

9. 密钥调度

密钥调度是负责把输入的密钥转化成所需的Nr+1个128位轮密钥。图4-11中的算法将计算轮密钥。

```
输入：
    Nk  密钥中的32位字的个数（4、6或者8）
    w   4×（Nr+1）个32位字的数组
输出：
    w   使用密钥来设置一个数组

  1. 以big-endian的格式把秘密密钥预加载到w的第一个Nk字中
  2. i = Nk
  3. while (i < 4*(Nr+1)) do
     1. temp = w[i - 1]
     2. if (i mod Nk = 0)
        i. temp = SubWord(RotWord(temp)) XOR Rcon[i/Nk]
  3. else if (Nk > 6 and i mod Nk = 4)
        i. temp = SubWord(temp)
  4. w[i] = w[i-Nk] xor temp
  5. i = i + 1
```

图4-11 AES密钥调度

密钥调度还需要两个额外的函数。SubWord()的输入为32位并且并行地把每个字节发送给AES的SubBytes替代表中。RotWord()把字向右循环移动8位。Rcon表是一个数组，它只存储了多项式$g(x)=x$的前10个幂模AES多项式的最高字节。

4.2 实现

已经有许多针对各种平台的公开的AES实现。从最常见的参考来看，这些实现从用于32位

和64位桌面到针对微控制器的8位的实现。也有各种针对硬件环境的实现，它们主要用来针对速度或者安全性（抵抗侧信道（side channel）攻击）或者二者兼有的优化。理想情况下，最好使用一个事先经过测试的AES的实现，而不要用自己所写的。但是，有些情况下是要求自定义的实现的，因此理解怎样去实现它就相当重要了。

我们将把注意力放在一个相当简单的8位实现上，它适合于微控制器上的紧凑实现。接着，我们将关注传统的32位实现，它常见于各种软件包中，例如OpenSSL，GnuPG和LibTomCrypt。

4.2.1 一个8位的实现

我们的第一个实现是一个直接使用字节数组将标准翻译为C语言。现在，我们并不准备应用任何优化，以使得这些C代码能足够的清晰。这个代码能够在许多环境下很好地运行，因为它使用很少的代码和数据空间并且是工作于小的8位数据类型上的。但是在速度是关键问题的地方使用它并不理想，而且也不建议用于商业的应用程序中。

aes_small.c:

```
/* The AES Substitution Table */
static const unsigned char sbox[256] = {
0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5,
0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76,
<snip>
0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68,
0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16 };

/* The key schedule rcon table */
static const unsigned char Rcon[10] = {
0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80,  0x1B, 0x36 };
```

这两个表形成了常量表。第一个是把SubBytes函数实现为一个单独的表，称为sbox。第二个表是用于密钥调度的Rcon表，它是g(x)=x的前10个幂。

```
/* The *x function */
static unsigned char xtime(unsigned char x)
{
   if (x & 0x80) { return ((x<<1)^0x1B) & 0xFF; }
   return x<<1;
}
```

这个函数实现了MixColumns需要的xtime。一种可能的折衷方案是把它实现为一个单独的256个字节的表。这样可以避免异或、移动和分支，使得代码变得更快，但这是以使用许多固定的数据为代价的。

```
/* MixColumns:  Processes the entire block */
static void MixColumns(unsigned char *col)
{
  unsigned char tmp[4], xt[4];
  int x;

  for (x = 0; x < 4; x++, col += 4) {
     xt[0]  = xtime(col[0]);
     xt[1]  = xtime(col[1]);
     xt[2]  = xtime(col[2]);
     xt[3]  = xtime(col[3]);
```

```
        tmp[0] = xt[0] ^ xt[1] ^ col[1] ^ col[2] ^ col[3];
        tmp[1] = col[0] ^ xt[1] ^ xt[2] ^ col[2] ^ col[3];
        tmp[2] = col[0] ^ col[1] ^ xt[2] ^ xt[3] ^ col[3];
        tmp[3] = xt[0] ^ col[0] ^ col[1] ^ col[2] ^ xt[3];
        col[0] = tmp[0];
        col[1] = tmp[1];
        col[2] = tmp[2];
        col[3] = tmp[3];
     }
  }
```

这是我们前面所看到的MixColumns函数，不同的是它现在做了修改以工作于状态中的所有的16个字节。正如前面提到的，这个函数也有双缓冲（复制到tmp[ ]），而且可以通过优化来避免这样。我们也使用一个数组xt[ ]来保存xtime()输出的拷贝。因为它用了两次，所以抓住了它就能节约时间。但是，实际上我们并不需要这个数组。如果我们第一次加上所有的输出，那么xtime()就会产生结果，因此我们只需要额外的一个字节的存储空间。

```
/* ShiftRows:  Shifts the entire block */
static void ShiftRows(unsigned char *col)
{
   unsigned char t;

   /* 2nd row */
   t = col[1]; col[1] = col[5];  col[5] = col[9];
   col[9] = col[13]; col[13] = t;

   /* 3rd row */
   t = col[2]; col[2] = col[10]; col[10] = t;
   t = col[6]; col[6] = col[14]; col[14] = t;

   /* 4th row */
   t = col[15]; col[15] = col[11]; col[11] = col[7];
   col[7] = col[3]; col[3] = t;
}
```

这个函数实现了ShiftRows功能。它使用了一个单独的临时字节*t*来交换行中的值。第二行和第四行实际上是使用一个移位寄存器来实现的，而第三行是使用一对交换。

```
/* SubBytes */
static void SubBytes(unsigned char *col)
{
   int x;
   for (x = 0; x < 16; x++) {
      col[x] = sbox[col[x]];
   }
}
```

这个函数实现了SubBytes功能。它是相当简单的，没有什么太多需要优化的地方。

```
/* AddRoundKey */
static void AddRoundKey(unsigned char *col,
                        unsigned char *key, int round)
{
   int x;
   for (x = 0; x < 16; x++) {
       col[x] ^= key[(round<<4)+x];
```

```
        }
    }
```

这个函数实现了AddRoundKey功能。它从一个单独的字节数组中读入轮密钥，它最多有15×16=240个字节的大小。我们把轮数左移4位以模拟乘以16。

这个函数在字的大小比8位大的平台上可以优化，通过一次对多个密钥字节进行异或来实现。这是一个我们将会在32位的代码中看到的优化。

```
/* Encrypt a single block with Nr rounds (10, 12, 14) */
void AesEncrypt(unsigned char *blk, unsigned char *key, int Nr)
{
   int x;

   AddRoundKey(blk, key, 0);
   for (x = 1; x <= (Nr - 1); x++) {
     SubBytes(blk);
     ShiftRows(blk);
     MixColumns(blk);
     AddRoundKey(blk, key, x);
   }

   SubBytes(blk);
   ShiftRows(blk);
   AddRoundKey(blk, key, Nr);
}
```

这个函数对存储在*blk*中的分组进行加密，并在适当的地方使用存储在*key*中的已调度过的秘密密钥。用到的轮数存储在Nr中且必须为10、12或者14，这取决于秘密密钥的长度（分别为128、192、256位）。

这个AES的实现没有经过太多的优化，因为我们想用行动来展现AES中离散的元素。更加特别地是，在轮当中，我们还有一些离散的步骤。我们将在稍后看到，即使目标是8位的，我们也可以把SubBytes、ShiftRows和MixColumns整合成一步，它节约了双缓冲池、置换（ShiftRows）和查找。

```
/* Schedule a secret key for use.
 * outkey[] must be 16*15 bytes in size
 * Nk == number of 32-bit words in the key, e.g., 4, 6 or 8
 * Nr == number of rounds, e.g., 10, 12, 14
 */
void ScheduleKey(unsigned char *inkey,
                 unsigned char *outkey, int Nk, int Nr)
{
   unsigned char temp[4], t;
   int           x, i;

   /* copy the key */
   for (i = 0; i < (4*Nk); i++) {
      outkey[i] = inkey[i];
   }

   i = Nk;
   while (i < (4 * (Nr + 1))) {
      /* temp = w[i-1] */
      for (x = 0; x < 4; x++) temp[x] = outkey[((i-1)<<2) + x];
```

```

        if (i % Nk == 0) {
           /* RotWord() */
           t = temp[0]; temp[0] = temp[1];
           temp[1] = temp[2]; temp[2] = temp[3]; temp[3] = t;

           /* SubWord() */
           for (x = 0; x < 4; x++) {
               temp[x] = sbox[temp[x]];
           }
           temp[0] ^= Rcon[(i/Nk)-1];
        } else if (Nk > 6 && (i % Nk) == 4) {
           /* SubWord() */
           for (x = 0; x < 4; x++) {
               temp[x] = sbox[temp[x]];
           }
        }

        /* w[i] = w[i-Nk] xor temp */
        for (x = 0; x < 4; x++) {
           outkey[(i<<2)+x] = outkey[((i-Nk)<<2)+x] ^ temp[x];
        }
        ++i;
     }
  }
```

这个密钥调度是直接使用8位的数据类型把标准的AES密钥调度翻译成C语言的。我们不得不使用一个移动来模拟RotWords()，而且所有的加载和存储都是通过一个4步的循环来完成的。

最明显的优化是对每个密钥大小创建一个循环并且去掉求余（%）运算。在优化的密钥调度方案中，我们稍后将看到，一个密钥能够以大约1 000 AMD 64个或者更少的时钟周期来调度。一个单独的除法就花费了多达100个时钟周期，因此删去这个操作是一种不错的开始。

与在32位和64位平台上的AddRoundKey一起，我们将用完全的32位字代替8位字来实现密钥调度。这使得我们可以有效地实现RotWord()和32位的异或运算。

```
/** DEMO **/

#include <stdio.h>
int main(void)
{
  unsigned char blk[16], skey[15*16];
  int x, y;
  static const struct {
     int Nk, Nr;
     unsigned char key[32], pt[16], ct[16];
  } tests[] = {
    { 4, 10,
      { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
        0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f },
      { 0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77,
        0x88, 0x99, 0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0xff },
      { 0x69, 0xc4, 0xe0, 0xd8, 0x6a, 0x7b, 0x04, 0x30,
        0xd8, 0xcd, 0xb7, 0x80, 0x70, 0xb4, 0xc5, 0x5a }
    }, {
      6, 12,
      { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
```

```
         0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
         0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17 },
       { 0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77,
         0x88, 0x99, 0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0xff },
       { 0xdd, 0xa9, 0x7c, 0xa4, 0x86, 0x4c, 0xdf, 0xe0,
         0x6e, 0xaf, 0x70, 0xa0, 0xec, 0x0d, 0x71, 0x91 }
     }, {
       8, 14,
       { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
         0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
         0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
         0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f },
       { 0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77,
         0x88, 0x99, 0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0xff },
       { 0x8e, 0xa2, 0xb7, 0xca, 0x51, 0x67, 0x45, 0xbf,
         0xea, 0xfc, 0x49, 0x90, 0x4b, 0x49, 0x60, 0x89 }
     }
  };
```

这3个条目是标准的针对128、192和256位密钥的AES测试向量。

```
  for (x = 0; x < 3; x++) {
     ScheduleKey(tests[x].key, skey, tests[x].Nk, tests[x].Nr);

     for (y = 0; y < 16; y++) blk[y] = tests[x].pt[y];
     AesEncrypt(blk, skey, tests[x].Nr);
```

这里我们对明文进行加密（blk==pt），而且将测试其结果是否等于预期的密文。

来自安全研究人员的忠告

分组密码测试

对提供的明文进行多次加密，并且少解密一次来检查你是否能够得到期望的结果，这是一个测试分组密码实现的很好的方法。例如，对明文进行加密，然后对密文再进行999次的加密。接着，重复地对密文进行999次解密，把结果和预期的密文进行比较。

通常，预计算的表条目会有很小的偏差，而且它仍然允许固定的向量通过测试。虽然这不太可能，但是在某些分组密码中（例如CAST5），完全可能出现这种情况。

这个测试更加适用于那些表格是双射的一部分的设计，例如AES MDS变换。如果这些表格中有错误，那么，这个实现应该不能合适地对密文进行解码并产生不正确的输出。

AES处理的一部分就是打算用来提供以这种形式测试向量的。测试者只需要简单地重复加密*N*次并且检验输出是否和期望的值相符，而不需要*N*−1次解密。这可以抓到设计中的错误，而这些设计的元素并不一定是一个双射（例如在Feistel分组密码中）。

```
     for (y = 0; y < 16; y++) {
        if (blk[y] != tests[x].ct[y]) {
           printf("Byte %d differs in test %d\n", y, x);
           for (y = 0; y < 16; y++) printf("%02x ", blk[y]);
           printf("\n");
           return -1;
        }
     }
  }
```

```
 printf("AES passed\n");
 return 0;
}
```

这个实现将作为我们的参考实现。现在我们来考虑各种各样的优化。

4.2.2 优化的8位实现

我们可以从参考实现中删除一些热点（hotspot）。

1. 把xtime()实现为一个查表操作。
2. 在轮函数中将ShiftRows和MixColumns进行整合。
3. 删除双缓冲。

新的xtime表如下。

aes_small_opt.c:

```
static const unsigned char xtime[256] = {
0x00, 0x02, 0x04, 0x06, 0x08, 0x0a, 0x0c, 0x0e,
0x10, 0x12, 0x14, 0x16, 0x18, 0x1a, 0x1c, 0x1e,
0x20, 0x22, 0x24, 0x26, 0x28, 0x2a, 0x2c, 0x2e,
<snip>
0xcb, 0xc9, 0xcf, 0xcd, 0xc3, 0xc1, 0xc7, 0xc5,
0xfb, 0xf9, 0xff, 0xfd, 0xf3, 0xf1, 0xf7, 0xf5,
0xeb, 0xe9, 0xef, 0xed, 0xe3, 0xe1, 0xe7, 0xe5 };
```

这个查找表和旧的函数返回相同的结果。现在我们正在删除一个函数调用、分支以及一些简单的逻辑运算。

接着，我们把ShiftRows和MixColumns整合到一个函数中。

aes_small_opt.c:

```
static void ShiftMix(unsigned char *col, unsigned char *out)
{
   unsigned char xt;

#define STEP(i,j,k,l)                                              \
   out[0] = col[j] ^ col[k] ^ col[l];                              \
   out[1] = col[i] ^ col[k] ^ col[l];                              \
   out[2] = col[i] ^ col[j] ^ col[l];                              \
   out[3] = col[i] ^ col[j] ^ col[k];                              \
   xt = xtime[col[i]];  out[0] ^= xt; out[3] ^= xt;                \
   xt = xtime[col[j]];  out[0] ^= xt; out[1] ^= xt;                \
   xt = xtime[col[k]]; out[1] ^= xt; out[2] ^= xt;                 \
   xt = xtime[col[l]]; out[2] ^= xt; out[3] ^= xt;                 \
   out += 4;

   STEP(0,5,10,15);
   STEP(4,9,14,3);
   STEP(8,13,2,7);
   STEP(12,1,6,11);

#undef STEP
}
```

我们去掉了双缓冲*tmp*数组并且正在输出到一个不同的目的地中去。接着，我们删除*xt*数组并且用一个单独的无符号字符型变量来代替它。

整个函数已经被分解开来，这使得数组的索引更加快速。在各种处理器中（例如8051），通过常量来访问其内部RAM是一个非常快速的（1个时钟周期）操作。虽然这会使代码变得更庞大，但它的确达到了一种不错的对性能的促进作用。在8051系列处理器上，实现者应该把*tmp*和*blk*映射到IRAM空间中去。

传入STEP宏的索引来自AES分组，它们都有一定合理数量的偏移。回忆一下前面提到过的，我们是以列序为主来存储数值。如果没有ShiftRows，其选择类型将会是{0, 1, 2, 3}、{4, 5, 6, 7}，等等。在这里，我们通过对AES状态中的字节进行重命名来把ShiftRows函数合并到代码中。现在字节1变成了字节5（位置1、1，而不是1、0），字节2变成了字节10，如此等等。这给出了如下的选择类型{0, 5, 10, 15}、{4, 9, 14, 3}、{8, 13, 2, 7}和{12, 1, 6, 11}。

我们可以用下面的代码来完成这个循环。

```
for (x = 0; x < 16; x += 4) {
  STEP((x+0)&15,(x+5)&15,(x+10)&15,(x+15)&15);
}
```

如果编译器能够足够智能地对这个宏使用CSE的话，那么上面这段代码可以达到将近4倍的代码压缩程度。对于各种嵌入式的编译器来说，你可能需要把i、j、k和l声明为局部变量。例如：

```
for (x = 0; x < 16; x += 4) {
  int i, j, k, l;
  i = (x+0)&15; j = (x+5)&15; k = (x+10)&15; l = (x+15)&15);
  STEP(i, j, k, l)
}
```

现在，当宏被展开时，将会使用预计算的值。由这个改变所带来的是，我们现在需要新的SubBytes和AesEncrypt函数来调整间接的输出缓冲区。

aes_small_opt.c:

```
/* SubBytes */
static void SubBytes(unsigned char *col, unsigned char *out)
{
   int x;
   for (x = 0; x < 16; x++) {
      out[x] = sbox[col[x]];
   }
}

<snip>

/* Encrypt a signle block with Nr rounds (10, 12, 14) */
void AesEncrypt(unsigned char *blk, unsigned char *key, int Nr)
{
   int x;
   unsigned char tmp[16];

   AddRoundKey(blk, key, 0);
   for (x = 1; x <= (Nr - 1); x++) {
     SubBytes(blk, tmp);
     ShiftMix(tmp, blk);
     AddRoundKey(blk, key, x);
   }
```

```
      SubBytes(blk, blk);
      ShiftRows(blk);
      AddRoundKey(blk, key, Nr);
}
```

这里我们仍然使用一个双缓冲体系（类似于图形程序设计中的翻页），不同的是我们并不是不做什么事情就将结果直接复制回去。SubBytes把结果存储在局部变量*tmp*数组中，然后ShiftMix再把数据输出回*blk*中。

通过这些所有的改变，我们现在可以完全地删除MixColumns函数了。在x86处理器上，这种代码差别是相当细微的，优化的版本只多了298个字节的代码空间。显然，在更小的，兼容性更差的处理器上，它的代码也不会增加多少。但是，从性能的提升来看，这样做是很值得的。

虽然在这里没有提到，但是解密也可以进行同样的优化。建议如果有空间可以用来进行在GF(2)[x]/v(x)上和9、11、13及14的乘法时，可以分别通过256个字节的表来实现。这样会增加1 024个字节大小的代码，但是却彻底地提升了性能。

提示 当设计一个加密系统时，需要注意的是许多模式并不需要相应的解密模式。我们将在稍后的章节中看到，CMAC、CCM和GCM操作模式对于加密和解密来说，仅需要密码的加密方向。这使得我们可以完全忽略解密函数并且节约可观的代码空间。

密钥调度的改变

既然我们已把ShiftRows和MixColumns进行了合并，那么解密就成为一个问题。在AES的解密时，我们应该在InvMixColumns这一步之前执行AddRoundKey，但是，因为这个优化，只能把它放到后面去（从技术上来看，并不是这样。通过正确的置换，我们可以把AddRoundKey放在InvShiftRows之前）。但是，这里提到的方案可以完成快速的32位实现。如果令S代表AES分组，K代表轮密钥，C代表InvMixColumn矩阵，我们应该计算C(S+K)=CS+CK。但是，现在如果我们在后面才加上轮密钥的话，那么就先不计算CS+K。

这种解决方案是比较简单的。如果对除了第一个和最后一个之外所有的轮密钥应用InvMixColumn的话，我们可以把它放到每一轮的最后并且仍然以CS+CK结束。通过这一调整，解密实现就可以使用合适的ShiftMix()变形将ShiftRows和MixColumns在一步中完成。读者应该注意这个调整，因为它也出现在快速的32位实现中。

4.2.3 优化的32位实现

我们的32位优化实现是以可移植的C语言所写的，它具有很高的性能。它基于由Rijndael小组提供的标准参考代码，而且是公开的。为了让AES在32位软件中更快，我们不得不把SubBytes、ShiftRows和MixColumns合并到一个更短的操作序列中。我们使用重命名来完成ShiftRows，使用一个单独的4个表的集来一次实现SubBytes和MixColumns。

1. **预计算表**

对于我们的实现，所需要的第一件事就是5张表，其中4张是用于轮函数的，另外1张是用

于最后一个SubBytes的（也可以用于逆密钥调度）。

前4个表是SubBytes和MDS变换的列的乘积。

(1) Te0[x] = S(x) * [2, 1, 1, 3]

(2) Te1[x] = S(x) * [3, 2, 1, 1]

(3) Te2[x] = S(x) * [1, 3, 2, 1]

(4) Te3[x] = S(x) * [1, 1, 3, 2]

其中S(x)是SubBytes变换，乘积是一个1x1 * 1x4的矩阵运算。从这些表中，我们能够用如下的代码来计算SubBytes和MixColumns。

```
unsigned long SubMix(unsigned long x)
{
   return Te0[x&255] ^
          Te1[(x>>8)&255] ^
          Te2[(x>>16)&255] ^
          Te3[(x>>24)&255];
}
```

第5个表只是简单地把SubBytes函数重复4次，即Te4[x]=S(x) * [1,1,1,1]。

我们注意到了一个空间上的优化（在实现的安全性上面也有作用），这些表只是互相移动来得到的。例如，Te1[x]=RotWord(Te0[*x*])、Te2[*x*]=RotWord(Te1[*x*])，等等。这意味着我们能够在运行的时候计算Te1、Te2和Te3，而且可以节约3KB的内存空间（或者可能是缓存）。

在我们提供的代码中，我们把Te0和Te4完全列了出来。但是，我们使用SMALL_CODE来提供了如果需要的时候可以把Te1、Te2和Te3删除的功能。

aes_tab.c:

```
static const unsigned long TE0[256] = {
     0xc66363a5UL, 0xf87c7c84UL, 0xee777799UL, 0xf67b7b8dUL,
     0xfff2f20dUL, 0xd66b6bbdUL, 0xde6f6fb1UL, 0x91c5c554UL,
     0x60303050UL, 0x02010103UL, 0xce6767a9UL, 0x562b2b7dUL,
     0xe7fefe19UL, 0xb5d7d762UL, 0x4dababe6UL, 0xec76769aUL,
     0x8fcaca45UL, 0x1f82829dUL, 0x89c9c940UL, 0xfa7d7d87UL,
<snip>
     0x038c8c8fUL, 0x59a1a1f8UL, 0x09898980UL, 0x1a0d0d17UL,
     0x65bfbfdaUL, 0xd7e6e631UL, 0x844242c6UL, 0xd06868b8UL,
     0x824141c3UL, 0x299999b0UL, 0x5a2d2d77UL, 0x1e0f0f11UL,
     0x7bb0b0cbUL, 0xa85454fcUL, 0x6dbbbbd6UL, 0x2c16163aUL,
};

static const unsigned long Te4[256] = {
     0x63636363UL, 0x7c7c7c7cUL, 0x77777777UL, 0x7b7b7b7bUL,
     0xf2f2f2f2UL, 0x6b6b6b6bUL, 0x6f6f6f6fUL, 0xc5c5c5c5UL,
     0x30303030UL, 0x01010101UL, 0x67676767UL, 0x2b2b2b2bUL,
     0xfefefefeUL, 0xd7d7d7d7UL, 0xababababUL, 0x76767676UL,
<snip>
     0xcecececeUL, 0x55555555UL, 0x28282828UL, 0xdfdfdfdfUL,
     0x8c8c8c8cUL, 0xa1a1a1a1UL, 0x89898989UL, 0x0d0d0d0dUL,
     0xbfbfbfbfUL, 0xe6e6e6e6UL, 0x42424242UL, 0x68686868UL,
     0x41414141UL, 0x99999999UL, 0x2d2d2d2dUL, 0x0f0f0f0fUL,
     0xb0b0b0b0UL, 0x54545454UL, 0xbbbbbbbbUL, 0x16161616UL,
};
```

这两个表是我们的Te0和Te4表。注意，我们是用TE0（大写）来命名的，因为这样就能够使用宏（如下）来访问这些表。

```
#ifdef SMALL_CODE
#define Te0(x) TE0[x]
#define Te1(x) RORc(TE0[x], 8)
#define Te2(x) RORc(TE0[x], 16)
#define Te3(x) RORc(TE0[x], 24)
#define Te4_0 0x000000FF & Te4
#define Te4_1 0x0000FF00 & Te4
#define Te4_2 0x00FF0000 & Te4
#define Te4_3 0xFF000000 & Te4
#else
#define Te0(x) TE0[x]
#define Te1(x) TE1[x]
#define Te2(x) TE2[x]
#define Te3(x) TE3[x]
static const unsigned long TE1[256] = {
    0xa5c66363UL, 0x84f87c7cUL, 0x99ee7777UL, 0x8df67b7bUL,
    0x0dfff2f2UL, 0xbdd66b6bUL, 0xb1de6f6fUL, 0x5491c5c5UL,
    0x50603030UL, 0x03020101UL, 0xa9ce6767UL, 0x7d562b2bUL,
    0x19e7fefeUL, 0x62b5d7d7UL, 0xe64dababUL, 0x9aec7676UL,
<snip>
```

在这里我们可以看到4个表的定义。我们也使得很大的代码变形来把Te4分成4个表。这省去了提取所需字节时需要的逻辑与操作。

在小的代码变形中，我们没有包含TE1、TE2或者TE3，而是使用循环移位宏RORc（将在后面定义）来模拟所需要的表。我们也使用所需的逻辑与操作来构造4个Te4表。

2. 用于解密的表

对于解密模式，我们需要一个类似的5个表的集合，不同的是它们是用于加密的表的逆。

(1) $Td0[x] = S^{-1}(x) * [14, 9, 13, 12]$;

(2) $Td1[x] = S^{-1}(x) * [12, 14, 9, 13]$;

(3) $Td2[x] = S^{-1}(x) * [13, 12, 14, 9]$;

(4) $Td3[x] = S^{-1}(x) * [9, 13, 12, 14]$;

(5) $Td4[x] = S^{-1}(x) * [1, 1, 1, 1]$;

其中$S^{-1}(x)$是InvSubBytes，行矩阵是InvMixColumns的列。通过这些式子，我们能够使用前面的技术来构造InvSubMix()。

```
unsigned long InvSubMix(unsigned long x)
{
   return Td0[x&255] ^
          Td1[(x>>8)&255] ^
          Td2[(x>>16)&255] ^
          Td3[(x>>24)&255];
}
```

3. 宏

我们的AES代码使用了一系列的可移植的C宏代码来辅助使用数据类型。前两个宏是STORE32H和LOAD32H，是用于辅助把32位的值作为一个字节数组来存储和加载的。AES使用bigendian数据类型，如果我们只是简单地加载32位字，那么在大部分的平台上我们将不能得到正确的结果。第三个宏RORc完成向右循环移动一个指定的（非常量）位数。第四个也是最后一个宏byte是把一个32位字的第*n*个字节提取出来。

aes_large.c:
```
/* Helpful macros */
#define STORE32H(x, y)                                   \
      { (y)[0] = (unsigned char)(((x)>>24)&255);    \
        (y)[1] = (unsigned char)(((x)>>16)&255);    \
        (y)[2] = (unsigned char)(((x)>>8)&255);     \
        (y)[3] = (unsigned char)((x)&255); }

#define LOAD32H(x, y)                                    \
      { x = ((unsigned long)((y)[0] & 255)<<24) | \
            ((unsigned long)((y)[1] & 255)<<16) | \
            ((unsigned long)((y)[2] & 255)<<8)  | \
            ((unsigned long)((y)[3] & 255)); }

#define RORc(x, y) \
((((unsigned long)(x)&0xFFFFFFFFUL)>> \
(unsigned long)((y)&31)) | \
((unsigned long)(x)<< \
(unsigned long)(32-((y)&31)))) & 0xFFFFFFFFUL)

#define byte(x, n) (((x) >> (8 * (n))) & 255)
```

这些宏对于大部分的密码学函数来说都是常用的，因此可以把它们放到你的加密算法源代码库是的一个通用头文件中去。这些宏实际上都移植于LibTomCrypt包中的宏。LibTomCrypt比这个有点更加高级，主要在于它可以自动地检测出各种平台，并且使用相应的快速的宏（例如，在x86处理器上加载littleendian字）。

在ARM（或者类似的）处理器系列上，byte()函数并不是很有效的。ARM7（我们所选择的平台）能够把byte加载并存储到32位寄存器中去。上面的宏在littleendian平台上可以安全地改成如下形式。

```
#define byte(x, n) (unsigned long)((unsigned char *)&x)[n]
```

在big-endian平台上，要将[n]替换成[3−n]。

4. 密钥调度

我们的密钥调度利用了这样一个事实的优点，即你能够很容易地把一个循环进行分解。我们仍然以（至少）32位的数据类型来执行所有的操作。

aes_large.c:
```
static unsigned long setup_mix(unsigned long temp)
{
   return (Te4_3[byte(temp, 2)]) |
          (Te4_2[byte(temp, 1)]) |
          (Te4_1[byte(temp, 0)]) |
```

```
            (Te4_0[byte(temp, 3)]);
}
```

这计算了密钥调度中的SubWord()函数。它并行地对temp中的字节使用SubBytes函数。Te4_n数组的值来自于Te4数组中除了第n个值被掩盖之外的所有的值。例如，Te4_3中的所有的字仅仅是最高的8个位非零。

这个函数也通过对temp中的字节进行重命名来对输入执行RotWord（）操作。注意，例如，实际上到Te4_3中去的字节是输入的第三个字节（和第四个字节相反）。

```

void ScheduleKey(const unsigned char *key, int keylen,
                       unsigned long *skey)
{
    int i, j;
    unsigned long temp, *rk;
```

这个函数有两个地方不同于8位的实现。首先，我们以字节类型传递密钥长度（keylen），而不是32位字。也就是说，有效的keylen值是16、24和32。第二个区别是，输出是存储在一个15×4个字的数组中而不是15×16个字节的。

```
    /* setup the forward key */
    i                 = 0;
    rk                = skey;
    LOAD32H(rk[0], key     );
    LOAD32H(rk[1], key +  4);
    LOAD32H(rk[2], key +  8);
    LOAD32H(rk[3], key + 12);
```

我们总是不管实际密钥大小是多少而只加载密钥的前128位。

```
    if (keylen == 16) {
        j = 44;
        for (;;) {
            temp  = rk[3];
            rk[4] = rk[0] ^ setup_mix(temp) ^ rcon[i];
            rk[5] = rk[1] ^ rk[4];
            rk[6] = rk[2] ^ rk[5];
            rk[7] = rk[3] ^ rk[6];
            if (++i == 10) {
               break;
            }
            rk += 4;
        }
```

这个循环是针对128位的密钥模式计算轮密钥。它被完全的分解开来，以便于每次迭代都能产生一个轮密钥并且完全避免了除法运算。

```
    } else if (keylen == 24) {
        j = 52;
        LOAD32H(rk[4], key + 16);
        LOAD32H(rk[5], key + 20);
        for (;;) {
            temp = rk[5];
            rk[ 6] = rk[ 0] ^ setup_mix(temp) ^ rcon[i];
            rk[ 7] = rk[ 1] ^ rk[ 6];
```

```
            rk[ 8] = rk[ 2] ^ rk[ 7];
            rk[ 9] = rk[ 3] ^ rk[ 8];
            if (++i == 8) {
                break;
            }
            rk[10] = rk[ 4] ^ rk[ 9];
            rk[11] = rk[ 5] ^ rk[10];
            rk += 6;
        }
    } else if (keylen == 32) {
        j = 60;
        LOAD32H(rk[4], key + 16);
        LOAD32H(rk[5], key + 20);
        LOAD32H(rk[6], key + 24);
        LOAD32H(rk[7], key + 28);
        for (;;) {
            temp = rk[7];
            rk[ 8] = rk[ 0] ^ setup_mix(temp) ^ rcon[i];
            rk[ 9] = rk[ 1] ^ rk[ 8];
            rk[10] = rk[ 2] ^ rk[ 9];
            rk[11] = rk[ 3] ^ rk[10];
            if (++i == 7) {
                break;
            }
            temp = rk[11];
            rk[12] = rk[ 4] ^ setup_mix(RORc(temp, 8));
            rk[13] = rk[ 5] ^ rk[12];
            rk[14] = rk[ 6] ^ rk[13];
            rk[15] = rk[ 7] ^ rk[14];
            rk += 8;
        }
    } else {
       /* this can't happen */
       return;
    }
}
```

最后两个分别计算192位和256位的轮密钥。此时，我们已经在*skey*数组中得到了所需要的轮密钥。我们稍后将看到计算用于解密模式的密钥。余下的部分AES代码实现了其加密模式。

提示 AES密钥调度实际上是为了能够在有限的存储空间的环境下有效地计算而设计的。例如，如果你观察用于128位密钥的密钥调度，会发现分解的循环只使用了rk[0...7]，其中rk[0...3]是当前的轮密钥，而rk[4...7]将会是下一轮的轮密钥。

实际上，密钥调度可以在合适的地方进行计算。例如，一旦我们覆盖掉rk[4]，我们也就不再需要rk[0]了。我们可以只让rk[4]等于rk[0]。对于rk[5,6,7]也是一样。这可以使得我们能够把密钥调度集成到加密过程中，并且只使用4×4=16个字节的内存空间，而不是默认的最小的11×4×4=176个字节。

同样的技巧可以用于192位和256位的密钥调度。

```
void AesEncrypt(const unsigned char *pt,
                      unsigned char *ct,
                      unsigned long *skey, int Nr)
{
```

再一次不同于8位的代码，其不同之处在于我们从*pt*数组中读入明文并且把密文存储到*ct*数组中。这个实现允许pt==ct，因此如果调用函数选择的话，它们是可以重复的。

```
        unsigned long s0, s1, s2, s3, t0, t1, t2, t3, *rk;
        int r;

        rk = skey;

        /*
         * map byte array block to cipher state
         * and add initial round key:
         */
        LOAD32H(s0, pt      ); s0 ^= rk[0];
        LOAD32H(s1, pt  +  4); s1 ^= rk[1];
        LOAD32H(s2, pt  +  8); s2 ^= rk[2];
        LOAD32H(s3, pt  + 12); s3 ^= rk[3];
```

这里我们把分组加载到数组[s0, s1, s2, s3]中，并且同时应用第一个AddRoundKey。

```
        /*
         * Nr - 1 full rounds:
         */
        r = Nr >> 1;
        for (;;) {
            t0 =
                Te0(byte(s0, 3)) ^
                Te1(byte(s1, 2)) ^
                Te2(byte(s2, 1)) ^
                Te3(byte(s3, 0)) ^
                rk[4];
            t1 =
                Te0(byte(s1, 3)) ^
                Te1(byte(s2, 2)) ^
                Te2(byte(s3, 1)) ^
                Te3(byte(s0, 0)) ^
                rk[5];
            t2 =
                Te0(byte(s2, 3)) ^
                Te1(byte(s3, 2)) ^
                Te2(byte(s0, 1)) ^
                Te3(byte(s1, 0)) ^
                rk[6];
            t3 =
                Te0(byte(s3, 3)) ^
                Te1(byte(s0, 2)) ^
                Te2(byte(s1, 1)) ^
                Te3(byte(s2, 0)) ^
                rk[7];
```

这是一个完整的AES轮函数。我们把[s0, s1, s2, s3]中的函数加密到集合[t0, t1, t2, t3]中去。我们可以清楚地看到4种以4个Te数组查找和异或的形式的SubBytes以及MixColumns的应用。

ShiftRows函数是通过使用重命名来完成的。例如，第一个输出（对于t0）是s0的字节3（AES输入的字节0），s1的字节2（AES输入的字节5），等等。下一个输出（对于t1）以同样的类型，只不过移动了4列。

这种同样类型的轮函数也是我们将要用于解密中的。我们也会遇到和优化了的8位的代码相

同的麻烦，即我们需要改变轮密钥以便于能够在MixColumns之后应用它并且产生正确的结果。

```
        rk += 8;
        if (--r == 0) {
            break;
        }
```

这个“break”使得我们在达到最后一轮的时候能够退出循环。回忆一下，AES有9、11或者13轮。这个循环会执行Nr/2−1次，这意味着我们不得不在循环的中间就退出，而不是在结尾。

```

        s0 =
            Te0(byte(t0, 3)) ^
            Te1(byte(t1, 2)) ^
            Te2(byte(t2, 1)) ^
            Te3(byte(t3, 0)) ^
            rk[0];
        s1 =
            Te0(byte(t1, 3)) ^
            Te1(byte(t2, 2)) ^
            Te2(byte(t3, 1)) ^
            Te3(byte(t0, 0)) ^
            rk[1];
        s2 =
            Te0(byte(t2, 3)) ^
            Te1(byte(t3, 2)) ^
            Te2(byte(t0, 1)) ^
            Te3(byte(t1, 0)) ^
            rk[2];
        s3 =
            Te0(byte(t3, 3)) ^
            Te1(byte(t0, 2)) ^
            Te2(byte(t1, 1)) ^
            Te3(byte(t2, 0)) ^
            rk[3];
```

这段代码处理偶数轮。我们使用[t0, t1, t2, t3]作为源并且将结果送回[s0, s1, s2, s3]。读者也许会注意到我们是用rk[0, 1, 2, 3]来作为轮密钥的。这是因为我们是从0偏移进入循环的，而应该是4（第一个AddRoundKey）。因此，循环的前一半使用rk[4, 5, 6, 7]，而后一半使用更低的字。

对这段代码进行折叠的一种简单的方法是，只使用第一个循环并且以如下的代码来结束每一次迭代。

```
s0 = t0; s1 = t1; s2 = t2; s3 = t3;
```

这使得加密模式的实现以很小的速度代价换取将近一半的大小。

```
    }

    /*
     * apply last round and
     * map cipher state to byte array block:
     */
    s0 =
        (Te4_3[byte(t0, 3)]) ^
        (Te4_2[byte(t1, 2)]) ^
```

```
            (Te4_1[byte(t2, 1)]) ^
            (Te4_0[byte(t3, 0)]) ^
            rk[0];
        STORE32H(s0, ct);
        s1 =
            (Te4_3[byte(t1, 3)]) ^
            (Te4_2[byte(t2, 2)]) ^
            (Te4_1[byte(t3, 1)]) ^
            (Te4_0[byte(t0, 0)]) ^
            rk[1];
        STORE32H(s1, ct+4);
        s2 =
            (Te4_3[byte(t2, 3)]) ^
            (Te4_2[byte(t3, 2)]) ^
            (Te4_1[byte(t0, 1)]) ^
            (Te4_0[byte(t1, 0)]) ^
            rk[2];
        STORE32H(s2, ct+8);
        s3 =
            (Te4_3[byte(t3, 3)]) ^
            (Te4_2[byte(t0, 2)]) ^
            (Te4_1[byte(t1, 1)]) ^
            (Te4_0[byte(t2, 0)]) ^
            rk[3];
        STORE32H(s3, ct+12);
    }
```

现在我们对分组应用最后的SubBytes、ShiftRows和AddRoundKey。我们以big-endian格式把输出存储到*ct*数组中。

5. 性能

以各种编译器来计算，包括GNU C和Intel C编译器，这个代码处理每个分组都会花费相当数量的时钟周期（如表4-1所示）。

表4-1 AES在各种处理器上的比较（GCC 4.1.1）

处理器	加密每个分组所需要的时钟周期[128位的密钥]
AMD Opteron	247
Intel Pentium 540J	450
Intel Pentium M	396
ARM7TDMI	3300（在一个Nintendo GameBoy上面进行度量的，它包含一个16MHz的ARM7TDMI处理器。我们把AES代码放在其内部的快速内存中（IWRAM）并且从在那里运行）
ARM7TDMI + 字节修改	

即使这个代码表现的还不错，但它还不是最好的。一些商用的实现正式的宣称，在Intel Pentium 4处理器上每个字节最多14个时钟周期（每个分组需要224个）。这个数据看起来很难达到，因为AES-128在其执行路径上至少要有420个操作码。每个分组224个时钟周期的结果意味着1个指令大概使用1.9个时钟周期，而人们从来没有听说过这种处理器。

6. x86的性能

由于添加了8个新的通用寄存器，AMD Opteron实现了一种不错的提高。如果研究GCC在

x86_64和x86_32平台上的输出，我们就可以发现这两者之间有一种不错的区别（如表4-2所示）。

表4-2　一个AES轮的前1/4

x86_64		x86_32	
movq	%r10, %rdx	**movl**	**4(%esp), %eax**
movq	%rbp, %rax	**movl**	**(%esp), %edx**
shrq	$24, %rdx	**movl**	**(%esp), %ebx**
shrq	$16, %rax	shrl	$16, %eax
andl	$255, %edx	shrl	$24, %edx
andl	$255, %eax	andl	$255, %eax
movq	TE1(,%rax,8), %r8	movl	TE1(,%eax,4), %edi
movzbq	%bl,%rax	movzbl	%cl,%eax
xorq	TE0(,%rdx,8), %r8	xorl	TE0(,%edx,4), %edi
xorq	TE3(,%rax,8), %r8	xorl	TE3(,%eax,4), %edi
movq	%r11, %rax	**movl**	**8(%esp), %eax**
movzbl	%ah, %edx	movzbl	%ah, %edx
movq	(%rdi), %rax	movl	(%esi), %eax
xorq	TE2(,%rdx,8), %rax	xorl	TE2(,%edx,4), %eax
movq	%rbp, %rdx	**movl**	**4(%esp), %edx**
shrq	$24, %rdx	shrl	$24, %edx
andl	$255, %edx	xorl	%eax, %edi
xorq	%rax, %r8		

这两段代码都实现了循环中的第一轮的第一个MixColumns步骤。要注意的是，编译器也在第一个MixColumns中调度了第二个的部分以达到更高的并行性。即使在表4.2中，x86_64看起来更长，但是它执行起来更快，部分原因是，因为它要在同样的时间更多地处理第二个MixColumns并且很好地利用了额外的寄存器。

在x86_32中，我们可以清楚地看到各种对栈的操作（黑体）。那些中的每一个在AMD处理器上都会花费我们3个时钟周期（最小）（大多数的Intel处理器上是2个时钟周期）。在轮函数的主循环中，64位的代码被编译成没有一个栈操作。而32位的代码在每一轮中大约有15个栈操作，这导致了每一轮至少要有45个时钟周期，或者全部的9轮需要花费405个时钟周期。

当然，我们并没有看到全部的405个时钟周期的损失，因为在同一时刻不止有一个操作码在执行。这个损失也被同样在关键路径上的并行负载（例如，来自Te表或者轮密钥中的负载）所掩盖。这些延迟以任何方式发生着，因此，实际上我们同时也在加载（或者存储到）栈并没有增加时钟周期的数量。

对于这两种情况，我们可以把GCC（本例中为4.1.1）产生的代码进行改进。在64位代码中，我们可以看到一对“shrq $24,%rdx”和“andl $255,%edx”。实际上并不需要*andl*操作，因为只是%rdx的低32位可能包含一些数值。对于9轮来说，这大概会节省36个时钟周期（取决于*andl*操作是怎样和其他的操作码相配合的）。

在32位代码中，从“*%esp*”中的两个加载（第二行和第三行）导致了一个根本不需要的3个周期的损失。在使用AMD Athlon（和Opterons）的情况下，加载存储单元会缩短加载操作（在某种环境下），但是加载总是至少花费3个时钟周期。把第二个加载改成“*movl %edx,%ebx*”意味着我们停止等待*%edx*，但损失只是一个时钟周期，而不是3个。这种改变会至少从9轮中节省$9 \times 2 \times 4 = 72$个时钟周期。

7. ARM的性能

在ARM平台上，我们不能把内存访问操作和其他的操作混合在一起，但是在x86平台上就可以。默认的byte()实际上相当地慢，至少对于ARM7来说使用GCC4.1.1时是这样的。为了编译轮函数，GCC试图把所有的1/4轮一次执行完。实际的代码列表是相当长的，但是，通过一些巧妙的处理，我们可以让源代码大约为1/4轮。

GCC是如此的灵巧，这足够奇怪了。第一次尝试就把除了第一个1/4轮之外所有的代码都注释掉了。GCC正确的检测到这是一个死循环并且把函数优化成一个简单的

```
.L2: b .L2
```

它在自身中无限地循环下去。第二次尝试是把如下的代码放到循环当中。

```
if (--r) break;
```

GCC再一次地对这个进行了优化，因为源代码中的变量s0、s1、s2和s3并没有被修改。因此，我们简单地把t0复制给它们，并且得到了如下的代码，它确实是用于1/4轮的。

```
mov     r3, lr, lsr #16
ldr     lr, [sp, #32]
mov     r0, r0, lsr #24
ldr     r2, [lr, r0, asl #2]
ldr     r0, [sp, #36]
mov     r1, r4, lsr #8
and     r3, r3, #255
ldr     lr, [r0, r3, asl #2]
and     ip, r5, #255
and     r1, r1, #255
ldr     r0, [r8, ip, asl #2]
ldr     r3, [r7, r1, asl #2]
eor     r2, r2, lr
eor     r2, r2, r0
eor     r3, fp, r3
eor     sl, r2, r3
```

这是一个AES的1/4轮，它使用了我们前面所提到的字节码的优化。

```
ldrb    r1, [r5, #0]
ldrb    r2, [sp, #43]
ldrb    ip, [r9, #0]
ldr     r0, [r6, r1, asl #2]
ldr     r3, [r7, r2, asl #2]
ldrb    r2, [sp, #33]
ldr     r1, [r4, ip, asl #2]
eor     r3, r3, r0
ldr     ip, [r8, r2, asl #2]
eor     r3, r3, r1
ldr     r2, [lr, #32]
eor     r3, r3, ip
eor     r3, r3, r2
```

我们能够看到，编译器简单地使用“装入字节”*ldrb*指令来分离32位字中的字节以得到表中的索引。因为它是ARM代码，它能够使用内联的“*asl #2*”，这可以把索引乘以4来访问表。大体上，这个优化代码每1/4轮大概减少了3个操作码。那么，为什么它可以更快？考虑每轮的内存操作数（如表4-3所示）。

表4-3 使用ARM7 AES代码的内存操作

	加载	存储	总共的内存操作
通常的C代码	30	65	95
优化的C代码	36	6	42

即使我们把数据放入GameBoy（我们的测试平台）的快速的32位内部内存中，仍然需要许多的时钟周期来访问它。根据各种讨论这个平台的非官方的数据，如果访问不是连续的，那么一个加载需要3个时钟周期，一个存储需要2个。在优化的代码中，仅内存操作就花费了每轮100个时钟周期，在整个译码过程中就占用了1000个时钟周期。

从理论上说，这个代码可以通过在移动到下一个的时候使用每个源字来变得更快。在我们的参考代码中，我们一次计算了轮函数当中的所有的32位字，这是通过读取一行中的其他4个字的字节来完成的。例如，考虑这个1/4轮。

```
t0 = rk[4];
t1 = rk[5];
t2 = rk[6];
t3 = rk[7];
t1 ^= Te3(byte(s0,0));
t2 ^= Te2(byte(s0,1));
t3 ^= Te1(byte(s0,2));
t0 ^= Te0(byte(s0,3));
```

理想情况下，编译器使用ARM处理器来代表*t*0、*t*1、*t*2和*t*3。用这种办法，我们可以连续地对访问字中的字节。下一个1/4轮将按照顺序使用*s*1的字节。用这种方式，所有的内存访问都是连续的了。

8. 小变体的性能

现在我们来考虑使用更小的表以及一个分解了的加密函数的性能。这个代码可以用于代码空间是比较珍贵的地方，而且用于如ARM系列的处理器上时表现得非常好（如表4-4所示）。

表4-4 在AMD Opteron平台上对使用小的和大的AES代码的比较

模　式	加密每个分组所需要的时钟周期（128位的密钥）
大的代码	247
小的代码	325

一个有趣的事需要指出来，那就是GCC是怎样对待循环移位操作的。在aes_large.c代码中，如果SMALL_CODE符号被定义了，我们就会得到如下的1/4轮的代码。

```
aes_large.s:
      movq    %rbp, %rax
      movq    %rdi, %rcx
      shrq    $16, %rax
      andl    $255, %eax
      movq    TE0(,%rax,8), %rdx
      movzbl  %ch, %eax
      movq    TE0(,%rax,8), %rsi
      movzbq  %bl,%rax
```

PDG

```
movq    TE0(,%rax,8), %rcx
movq    %r11, %rax
movq    %rdx, %r10
shrq    $24, %rax
salq    $24, %rdx
andl    $4294967295, %r10d
andl    $255, %eax
shrq    $8, %r10
orq     %rdx, %r10
andl    $4294967295, %r10d
xorq    TE0(,%rax,8), %r10
movq    %rcx, %rax
andl    $4294967295, %eax
salq    $8, %rcx
shrq    $24, %rax
orq     %rcx, %rax
andl    $4294967295, %eax
xorq    %rax, %r10
```

正如我们所看到的，GCC在使用一个64的数据类型来做一个32位的循环移位操作。同样的代码以32位的模式进行编译将得到如下的1/4轮的代码。

aes_large.s (32-bit):

```
movl    4(%esp), %eax
movl    (%esp), %ebx
shrl    $16, %eax
shrl    $24, %ebx
andl    $255, %eax
movl    TE0(,%eax,4), %esi
movzbl  %cl,%eax
movl    TE0(,%eax,4), %eax
rorl    $8, %esi
xorl    TE0(,%ebx,4), %esi
movl    %ebp, %ebx
rorl    $24, %eax
xorl    %eax, %esi
movzbl  %bh, %eax
movl    4(%esp), %ebx
movl    TE0(,%eax,4), %eax
shrl    $24, %ebx
rorl    $16, %eax
xorl    (%edx), %eax
xorl    %eax, %esi
```

我们可以清楚地看到GCC识别出了循环移位操作，并且使用了一个不错的rorl指令来代替所有的移位、与（AND）和或（OR）运算。对于64位的情况来说，这种方案是很简单的，即使用一个32位的数据类型。至少对于GCC来说，符号__x86_64__是针对64位的x86_64模式而定义的。如果我们把下面的代码插到顶部并且用ulong32来代替所有的unsigneded long，我们就可以即支持32位也支持64位模式。

```
#if defined(__x86_64__) || (defined(__sparc__) && defined(__arch64__))
   typedef unsigned ulong32;
#else
   typedef unsigned long ulong32;
#endif
```

我们也加入了一个针对SPARC机器的通用组合定义来让代码更加整洁。现在，通过在合适的地方进行替代，我们可以看到GCC对我们的代码进行了很好的时间优化。

aes_large_mod.s:

```
movl    %ebp, %eax
movl    %edi, %edx
movq    %rdi, %rcx
shrl    $16, %eax
shrl    $24, %edx
movzbl  %al, %eax
movzbl  %dl, %edx
movq    TE0(,%rax,8), %rax
movl    %eax, %r8d
movzbl  %bl, %eax
rorl    $8, %r8d
movq    TE0(,%rax,8), %rax
xorl    TE0(,%rdx,8), %r8d
movq    %r11, %rdx
rorl    $24, %eax
xorl    %eax, %r8d
movzbl  %dh, %eax
movl    %ebp, %edx
movq    TE0(,%rax,8), %rax
shrl    $24, %edx
movzbl  %dl, %edx
rorl    $16, %eax
xorl    (%r10), %eax
xorl    %eax, %r8d
```

来自安全研究人员的忠告

了解你的数据类型

我们在aes_large.c的例子中看到，使用错误的数据类型会导致代码的执行效率很低。那么，怎样知道会在什么时候陷入这个问题呢?

对于初学者来说，比较好的做法是了解你的平台和参考C代码相比较会怎样。比如，“unsigned long”至少是32位长。它并不需要这么短，实际上，在大多数的64位平台上它是64位长。

另一种了解代码是否使用了错误类型的办法是检查汇编输出。通过使用-S选项来调用GCC（举例），编译器会产生你能够针对性能因素而进行的审计和检查的汇编。一种可以正确表明你正在使用错误类型的迹象是，你是否正在使用内部辅助程序（在一个32位的目标平台上，对64位的值是有用的）。

但是，内部函数并不总是可以用于这种判断，因为GCC是足够灵巧的，以致于它会内联许多简单的操作，比如在一个32位的平台上会使用64位的加法。在我们的这个例子中，注意到GCC使用移位、或和与运算来实现它能够单独地用x86操作码。

9. 逆密钥调度

至此，我们仅考虑了前面模式。至少是对这种实现方式来说，其逆算法看起来确实和正向的算法是相同的。关键的区别在于我们用前向表的逆表来代替它们，这就要求我们要看一下密

钥调度算法。

在标准的AES参考代码中，密钥调度算法对逆算法没有什么改变，因为我们必须在InvMixColumns之前应用AddRoundKey。但是，在这种实现方式中，我们在一个单独的短的操作序列中就执行了大部分的轮函数。我们无法在这些代码之间插入AddRoundKey，因此我们必须把它放到后面去。

其方案是对前向轮密钥执行如下的两步操作。

(1) 逆转轮密钥数组。

- 把轮密钥组合成几个128位的字；
- 把这些128位的字也进行逆转；
- 把轮密钥拆分回128位的字。

(2) 对除了第一和最后一个轮密钥之外的所有轮密钥应用InvMixColumns。

为了把密钥逆转，我们实际并不需要真的形成128位的字，而逻辑上是对32位字进行交换。下面的C代码会把*rk*中的密钥逆转到*drk*中。

```
rk += 10*4; /* last 128-bit round key for AES-128 */
for (x = 0; x < 11; x++) {
   drk[0] = rk[0];
   drk[1] = rk[1];
   drk[2] = rk[2];
   drk[3] = rk[3];
   rk -= 4; drk += 4;
}
```

现在在*drk*数组中，我们有了以相反的顺序存储的密钥。接着，我们要应用InvMixColumns。到现在我们还没有一种更好的方法来实现这个函数。表*Td*0、*Td*1、*Td*2和Td3实际上只是实现了InvSubBytes和InvMixColumns。但是，我们确实有一个SubBytes表是可以用到的（Te4）。如果我们首先把密钥的字节传入Te4，那么通过优化了的逆程序，我们就可以用

```
drk[x]    :=    InvMixColumns(InvSubWord(SubWord(drk[x]))
drk[x]    :=    InvMixColumns(drk[x])
```

来结束，这可以用下面的方法来实现。

```
for (x = 4; x < 10*4; x++) {
    drk[x] = Td0(255 & Te4[byte(drk[x], 3)]) ^
             Td1(255 & Te4[byte(drk[x], 2)]) ^
             Td2(255 & Te4[byte(drk[x], 1)]) ^
             Td3(255 & Te4[byte(drk[x], 0)]);
}
```

现在我们就有了用于AES-128的合理的逆密钥调度算法。对于192位和256位的来说，只要把“10*4”替换成“12*4”或者“14*4”就可以。

4.3 实用的攻击

在写作本书的时候，还没有已知的能攻破AES分组密码的数学设计原理的算法。也就是说，除了给定传统意义上的明文和密文的映射之外，不再给定其他信息的话，是没有一种能够比穷

举更快地找出密钥的办法的。

但是，这并不意味着AES是完美的。我们“经典的”32位的实现，虽然是很快并且高效的，但却泄露了许多侧信道数据。分别由Bernstein和Osvik（et al.）所开发的两种独立的攻击，使用了一种叫做侧信道攻击的方法来利用（exploit）这个实现。

4.3.1 侧信道

为了理解这些攻击，我们需要了解什么是侧信道。当我们运行AES密码时，产生了一个叫做密文的输出（或者是明文，这取决于加解密的方向）。但是，这个实现也产生其他的可以测量的信息。这个算法的执行并不花费固定的时候或者固定数量的能源。

从信息理论的意义上来说，它不需要固定的时间这个事实意味着这个实现会泄露关于算法内部状态的（以及它所在的运行设备）信息（熵）。如果攻击者能够把运行时和这个实现的知识联系起来的话，他能够，至少在理论上，提取关于这个密钥的信息。

那么，为什么我们的实现没有一个固定的执行时间呢？因为对于典型的处理器缓存来说，存在两个主要的可利用缺陷。

4.3.2 处理器缓存

处理器缓存是一个存储了最近所写的或者所读的数据的处理器，它独立于系统主存。虽然缓存以各种各样的形状和大小来设计，但都有一些使得它们可以被很容易地利用的典型的特征。缓存典型地具有低的组相联性并且使用存储体选择器（bank selector）。

1. 相联缓存（Associative Caches）

在一个典型的处理器缓存中，一个给定的物理（或者逻辑，这取决于其设计）地址需要映射到缓存中的某个位置。它们用称做缓存线（cache line）的内存单元来工作，其范围从小的16个字节到更为典型的64甚至128个字节。如果两条源地址（或者缓存线）映射到相同的缓存地址，那么它们中的一个不得不从缓存中被驱逐（evict）出去。驱逐（evition）意味着对于丢失的源地址，当下一次再使用它时，必须从内存中把它再取回来。

在一个全相联缓存（也叫做全相联存储器completely associated memory，叫CAM）中，一个源地址能够映射到缓存中的任何地方。这可以产生高的缓存命中率，而驱逐情况则会很少发生。这种类型的缓存是昂贵的（从耗费的空间方面来讲）并且实现起来很慢。增加缓存命中的延迟时间通常是不值得的，否则缓存中较少的保存会损失。

在一个组相联（set-associative）缓存中，一个给定的源地址能映射到N个惟一的位置中的一个，这也叫做N-路（N-way）相联缓存。这些缓存实现起来很经济，因为你只需要把一个要被驱逐的缓存线和缓存的N条路进行比较就可以。这降低了使用缓存的延迟，但会使缓存驱逐（cache eviction）更加可能。

在组相联模型中最经济地计算一个缓存地址的办法是，使用地址的连续位来作为一个缓存地址。这些常取自地址的中间部分，因为低的位是用于索引和存储选择的。这些细节一般来说在很大程度上取决于缓存的配置及其体系结构。

2. 缓存组成

缓存的组成会影响你能多么有效地访问它。在这一小节中，我们将考虑AMD Opteron 缓存设计［AMD64处理器的AMD软件优化指引（AMD Software Optimization Guide for AMD64 Processors），#25112，Rev 3.06，2005年9月］，但大部分的讨论也适用于其他的处理器。

AMD Opteron把地址分成几个部分：

(1) 索引（index）是地址的addr[14:6]。

(2) 存储体（bank）是地址的addr[5:3]。

(3) 字节（byte）是地址的addr[2:0]。

这些值取自L1缓存设计，它是一个64KB的两路组相联缓存。这个缓存实际上是在两条路中以两个32KB的数组所组成的（因此总共为64KB）。索引值选择所用的64字节的缓存线，存储体选择缓存线中的8字节组，字节表示了8字节组的起始位置。

这种缓存是双通信的，意思是在每个时钟周期中可以执行两次读操作，前提是没有发生存储体冲突（bank conflict）。处理器在并行总线上移动缓存中的数据。其意思是说所有的存储体0的事务发生在一个总线上，存储体1的事务在另一个总线上，等等。当两次具有不同索引的读操作来自同一个存储体时，冲突就会发生。也就是说，你是在从两个不同的缓存线中读取相同的存储体偏移。例如，使用AT&T语法，下面的代码将会产生一个存储体冲突。

```
movl (%eax),%ebx
movl 64(%eax),%ecx
```

假设这两个地址都在L1缓存中且%eax是以4个字节对齐的，这个代码将会产生一个存储体冲突。实际上，我们是想避免读取一个合理的64的倍数的偏移，但我们将看到对于我们的实现来说却是相当不幸的。

4.3.3 Bernstein 攻击

Bernstein第一个描述了针对AES的一种完全的缓存攻击（尽管这看起来很奇怪，但它的确是可行的），他使用了AES的实现及其运行的平台知识（他随后写了一篇后续论文，http://cr.yp.to/antiforgery/cachetiming-20050414.pdf）。他的攻击如下所示。

1. 取16个字节明文中的一个，称为pt[n]。

2. 遍历pt[n]所有的256种可能值并看哪一个加密所花的时间最长。

这种攻击基于这样一个事实，即pt[n] XOR key[n]将直接进入第一轮的表中。首先在一个已知密钥的情况下运行这个攻击。可以推导出一个值T= pt[n] XOR key[n]，它是最经常出现的。也就是说，你可能会看到T的其他值，但有一个值实际上是最经常出现的。现在，通过对你所选择的明文进行多次重复的加密，你可以对目标运行这个攻击。最后你会得到pt[n]的一个值，其延迟是最长的，然后你就能够推导出pt[n] XOR T = key[n]。

这种攻击看起来是不可行的，但实际上是如果没有它的这种流程，它就更不可能。不过，在足够长的运行之后，几乎对于所有的密钥字节，它都会产生一个稳定的T值。那么为什么这个攻击又是可行的？有两个原因，取决于环境因素，它们都能够成为真实的情形。

- 当一个缓存线被从缓存中驱逐出去时，会导致一个来自L2或者系统内存的加载；

• 一个（或者n个）存储体冲突会在第一轮中发生。

对于第一种情况，我们其中一个表会面临着缓存线驱逐的情况，因为另外的处理过程会竞争缓存线。由于另外的任务（像在Osvik攻击中的情况一样）或者内核处理程序对加密过程的中断，这种情况就会发生。大多数的处理器（包括带有小的8-16KB L1缓存的Pentium 4）能够把整个4KB的表放到L1缓存中。因此，如果OS（或者调用程序）并不驱逐它们的话，紧密的执行应该会很少发生缓存线驱逐。

这个攻击能够成功的最可能的原因是存储体冲突，即改变pt[n]的值直到它在相同的AES的1/4轮中和另一个查找发生冲突。每个表都是1024个字节，这是64的一个合理的倍数。这意味着，如果你把访问相同的32位字（或者它的邻居）作为另一次访问，那么将会导致延迟。

Bernstein针对这个问题提出了几种解决办法，最实用的一种方法是把所有的加载操作进行串行化。在AMD处理器上是3条整数流水线，每一条在一个时钟周期内都能产生一个加载或者存储操作（但处理起来最多需要两个时钟周期）。调整的策略是以3条指令为一组进行组合，并且要保证加载和存储都发生在组合中的相同位置。用各种长度的无操作（NOP）操作码来进行填充能够确保保持这种对齐方法。

但是这种调整也是最难应用的，因为它需要用汇编来实现。同时，它也是不可移植的，甚至在x86处理器平台上也是如此。甚至在某些单标量处理器上（例如ARM系列）它也是一个问题。

4.3.4 Osvik 攻击

Osvik（Dag Arne Osvik、Adi Shamir和Eran Tromer：*Cache Attacks and Countermeasures: the Case of AES*）攻击是一种比Bernstein攻击更加有效的攻击。在Bernstein攻击中只是简单地观察加密的时间，而在这种攻击中是试图把缓存中的东西驱逐出去以影响时间。

他们的这两种攻击和Colin Percival（Colin Percival: *Cache Missing For Fun and Profit*）攻击很相似，它是在目标机器上做进一步处理，增加特定的缓存线以期望能影响加密程序。其目的是，知道哪些线会被驱逐以及怎样把它们和密钥位联系起来。在他们的攻击中，能够在一分钟内读取密钥中的47位。

抵抗这种攻击比抵抗Bernstein攻击更为复杂。仅仅把加载进行强制性的串行是不够的，因为这不能避免缓存命不中的情况。在他们的论文中，他们提出了各种修改来减轻攻击；也就是说，不去抵抗这些攻击而是让这些攻击变得不可行。

4.3.5 挫败侧信道

这些侧信道究竟是不是一种威胁在很大程度上取决于你的应用程序的威胁模型及其用例。用一句概括的话来说，我们必须每时每刻都防卫所有的侧信道，实际上这是不明智的。首先，这么做不实际。其次，抵抗那些不算威胁的侧信道会花费很多的代价，而且通常你也没有时间去这样做。记住，你还要给客户提供产品。

这些攻击通常并不是那样实用的。例如，Bernstein攻击，虽然在理论上是有效的，但是它很难应用于实际，因此它需要具有能够对一个加密程序执行数百万次查询的能力，而且还要有

一个低延迟的信道以观察加密是怎样执行的。我们将看到通过使用经过认证的信道，那些从一个未经过认证的第三方而来的加密请求可以很容易地被避免。我们只要强迫请求者对他们的请求进行认证，并且如果认证失败，我们就终止会话（这样也可能阻止来自远程连接的攻击者）。攻击同样也需要一个低延迟的信道，因此也可以很好地和时间观察联系起来。仅仅是在实验室的环境中，这两台机器（攻击者和受害者）才能在相同的网络环境中。如果他们是通过互联网从一个传输媒介经过许多跳跃到另一个传输媒介的话，这就完全是另一种情况了。

Osvik攻击需要一个攻击者在目标机器上具有本地运行程序的能力。如果你正在运行一台服务器，那么最简单的办法就是不要允许用户在机器上拥有一个shell。

注意 这些攻击的意义已经引起了各种密码学流派之间的争论。但没有人争论这些攻击是否可行，这些攻击能否在实际的运行环境中有效地工作还有很多的问题。Bernstein的攻击需要一个应用程序来处理上万条未经授权的加密请求。Osvik的攻击需要一个攻击者能够在目标机器上运行程序。

处理这个信息最安全的办法是意识到它但不要特别害怕它。

从内核中得到的一些帮助

假设你已经排除了错误的威胁因素并且仍然认为缓存撞击攻击（cache smashing attack）是一个问题。这仍然存在一种可行的解决方案而不需要重新设计新的处理器或者其他不实际的解决办法。目前，虽然还没有项目提供这种解决办法，但还没有发现为什么不能这样做的真正原因。

以一个实现了一个串行AES的内核驱动（例如模块，设备驱动）开始并且在处理数据之前把表预加载到L1中。也就是说，所有对表的访问都被限制在一个特定的流水线（或者加载存储单元流水线）中。这对于处理器的流水线尤其特殊，而且对于不同的体系结构的处理器也有所不同。

接着，实现一个双缓冲机制，在其中，把要处理的文本加载到一个可以缓存的缓冲区中并且不会和AES表格相竞争。例如，在AMD处理器上你所要做的只是确保缓冲区不要和表模32768相重叠就可以了。假如这些表占用5KB，那么就留给我们27KB来存储数据。

现在对一个数据分组进行处理，内核把数据送去加密（或者解密）并且锁上机器（停止所有其他的处理器）。接着，处理器会尽可能多地对缓冲区进行填充（按照要求进行循环以填满该请求），把它预加载到L1，然后继续使用AES来处理数据（用一个合适的链接模式）。

这种办法能挫败Bernstein和Osvik这两种攻击，因为它没有存储体冲突且没有办法把缓存线驱逐出L1。很明显，这种办法会导致对主机的拒绝服务（DoS）问题，因为一个攻击者可以请求大量的加密来锁住机器。解决这种问题的办法是，需要某个用户的标识符来使用设备或者限制允许的数据的最大值。一个小到4～16KB的值足以让设备实际运行了。

4.4 链接模式

作为保密性的原型，分组密码本身并没有什么用处。用分组密码直接对数据进行加密是一种叫做电子密码本（Electronic Codebook, ECB）的模式。这种模式无法实现保密性，因为它会泄露明文的信息。一种好的链接模式的目标是能够比ECB模式提供更大的保密能力。好的链接

模式并不是用于认证的，这一点现在还不能过多地强调，但我们将在后面再次讨论它。

我们多次看到人们写的安全软件，虽然他们实现了保密性的原型，但无法确保其认证——这是由于他们不了解这些知识或者没有能力来实现它。更糟地是，人们使用像CBC这样的模式并且假设它也提供了认证能力。

ECB模式只所以无法完成保密是基于这样一个简单的事实，即如果相同的明文分组被多次加密的话它就会泄露信息。这在相同的会话（例如一个文件）中以及不同的会话中都会发生。例如，如果一台服务器用一个有限的用ECB模式加密的响应子集来响应一个查询，那么攻击者就只需要很小的映射空间就可以完成解码以了解服务器发出的响应。

为了更好地了解这一点，考虑一个信用卡认证消息。在它的最基本的层次上，响应要么是成功要么是失败，即0或者1。如果我们用ECB对这些消息进行加密，这并不需要攻击者花费多少工作就可以求出成功和失败的代码是什么了。

回到DES的年代，有许多流行的链接模式，密文反馈（Ciphertext Feedback，CFB），输出反馈（Output Feedback，OFB）以及密码分组链接（Cipher Block Chaining，CBC）。如今，最常见的模式是密码分组链接和计数器模式（Counter Mode，CTR）。它们是NIST SP 800-38A标准（以及OFB和CFB模式）的一部分，并且也用于其他协议（CTR模式用于GCM和CCM，CBC模式用于CCM和CMAC）。

4.4.1 密码分组链接

密码分组链接（CBC）模式是最常见的遗留下来的加密模式。它很容易理解并且在一个已有ECB模式的密码实现的基础上也可以很简单地实现。它常被误认为能提供认证，主要是因为“密文的变化将会使明文发生很大的变化”这个原因。只改变密文中的一个单独的位就会使得明文的两个分组产生很大的变化，这是真的。但它能提供认证功能不是真的。

在图4-12中，我们对明文分组P[0, 1, 2]进行加密，首先把分组和一个值进行异或（XOR），然后再加密。对于分组P[1, 2]，我们使用前一个密文C[0, 1]作为其链接值。由于第一个分组并没有前一个密文，那么我们怎样处理呢？为了解决这个问题，我们使用一个初始值（Initial Value，IV）来作为其链接值。然后消息就以密文分组C[0, 1, 2]和IV进行传输。

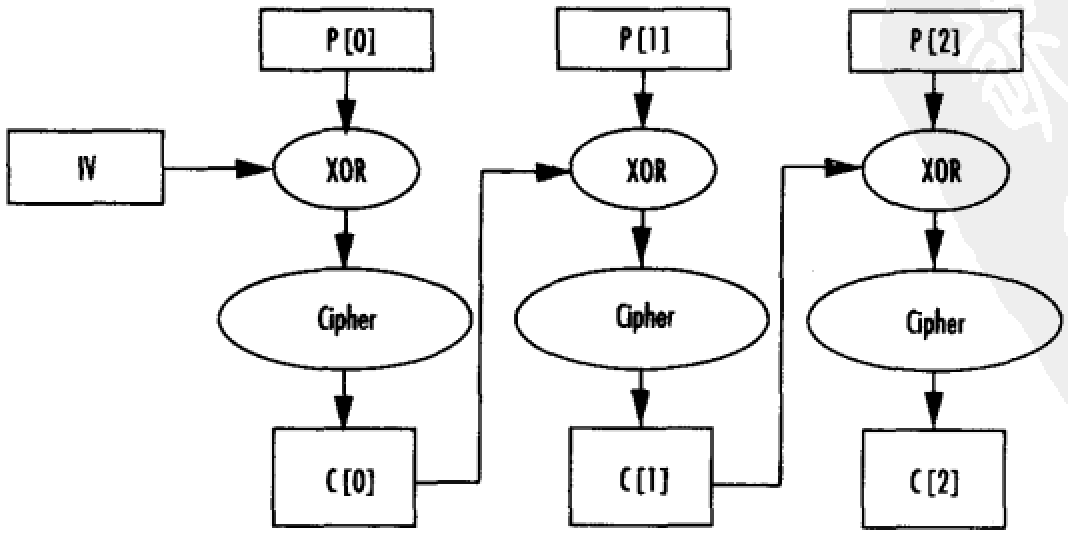


图4-12 密码分组链接模式

1. 什么是IV

一个初始值，至少对于CBC模式来说，必须随机选取并且需要保密。它必须和密码分组的长度相同（例如，对于AES可以为16个字节）。

有两种常用的办法来存储（或者传输）IV。最简单的办法是随机地生成IV并和消息一起存储。这使得消息的大小增加了一个单独的分组。

另外一种常用的处理IV的办法是在密钥协商处理的过程中生成它。使用一个称为PKCS#5的算法（参见第5章），我们能够看到从一个随机生成的共享秘密（shared secret）中，即能够得到加密密钥，也能够获得链接IV。

2. 消息长度

从示意图（图4-12）中可以看到，所有的消息是密码分组大小的某个倍数。实际上并不是这样。有两种常见的推荐解决办法（两个都可用）。不幸地是，FIPS 81（SP800-38A的指导条文）并没有给出一个解决方案。

第一种常用的办法是使用一种称为密文窃取（ciphertext stealing）的技术。在这个办法中，把最后一个密文分组传递给ECB模式的密码，并且把它的输出和剩余的消息字节（或位）进行异或。这避免了增加消息长度（除了添加IV之外）。

第二种方法是用足够的0位对最后一个分组进行填充，使得消息长度是分组大小的某个倍数。这增加了消息的大小而且和密文窃取相比并没有什么长处。

实际上，没有来自NIST的标准要求必须使用哪种方法而不能使用另外一种方法。最好是默认使用密文窃取，但是要标明你选择了这个模式，除非你非要坚持其他的标准。

3. 解密

用CBC模式进行解密，只需要把密文传给逆算法并把输出和链接值进行异或（如图4-13所示）。

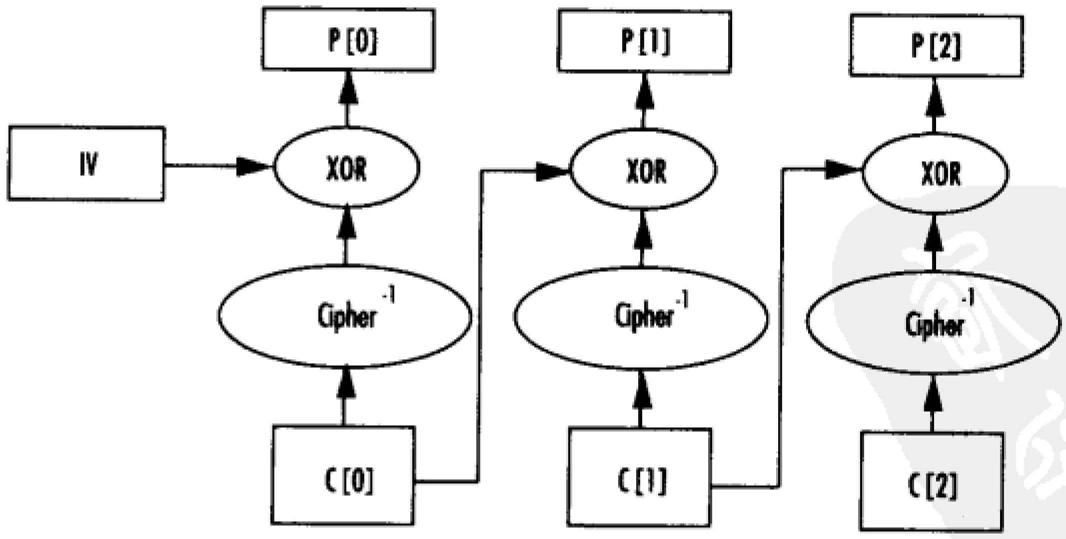


图4-13 密码分组链接的解密

4. 性能上的不足

CBC模式在加密过程中会碰到一个串行化的问题。密文C[n]只有在知道C[n−1]的情况下才能计算出来，这使得密码不能以并行的方式进行调用。解密是可以并行执行的，因为此时密文都已经知道。

5. 实现

把快速的32位AES作为一个模型，我们能够像下面一样实现CBC模式。

cbc.c:
```
009   void AesCBCEncrypt(const unsigned char *IV,
010                               unsigned char *pt,
011                               unsigned char *ct,
012                               unsigned long  size,
013                               unsigned long *skey, int Nr)
014   {
015      unsigned char buf[16];
016      unsigned long x;
017
018      for (x = 0; x < 16; x++) buf[x] = IV[x];
019      while (size--) {
020         /* create XOR of pt and chaining value */
021         for (x = 0; x < 16; x++) buf[x] ^= pt[x];
022
023         /* encrypt it */
024         AesEncrypt(buf, buf, skey, Nr);
025
026         /* copy it out */
027         for (x = 0; x < 16; x++) ct[x] = buf[x];
028
029         /* advance */
030         pt += 16; ct += 16;
031      }
032   }
```

这段代码以字节为基本实现了CBC模式。在实际应用中，一次对整个字进行异或会更快而且也可以移植。在x86处理器上，可以使用32位或者64位的字来执行异或操作（第21行代码）。这显著地提高了CBC代码的性能并且降低了原始的ECB模式的开销。

如果我们允许*pt*等于*ct*的话，那么解密就要稍微有点技巧了。

cbc.c:
```
034   void AesCBCDecrypt(const unsigned char *IV,
035                               unsigned char *ct,
036                               unsigned char *pt,
037                               unsigned long  size,
038                               unsigned long *skey, int Nr)
039   {
040      unsigned char buf[16], buf2[16], t;
041      unsigned long x;
042
043      for (x = 0; x < 16; x++) buf[x] = IV[x];
044      while (size--) {
045         /* decrypt it */
046         AesDecrypt(ct, buf2, skey, Nr);
047
048         /* copy current ct, create pt and then update buf */
049         for (x = 0; x < 16; x++) {
050            t      = ct[x];
051            pt[x]  = buf2[x] ^ buf[x];
052            buf[x] = t;
053         }
054
055         /* advance */
```

```
056         pt += 16; ct += 16;
057      }
058   }
```

这里我们再一次使用*buf*作为链接缓冲区。通过AES ECB模式把它解密到*buf*2中，因为此时我们还不能直接写到*pt*中。在其内部循环中（第49行～第53行代码），我们取出*ct*的一个字节，这是在覆盖*pt*中相当的位置之前进行的。这允许缓冲区能够重叠，而不用丢掉要用于CBC模式解密的前一个密文分组。

4.4.2 计数器模式

计数器模式是被设计用来提取分组密码的最大效能以实现保密性。在这种模式中，我们利用了这样一个事实，即分组密码应该是一个好的伪随机置换（PRP）。与把消息传递给分组密码进行加密不同的是，我们像用流密码加密一样对它进行加密。

在CTR模式中，有一个类似于CBC的IV，不同的是它并不需要是随机的，只要是惟一的就可以。在对IV进行加密之后，将输出和第一个明文分组进行异或以产生第一个密文。然后IV递增（increment）（好像它是一个大整数）且对下一个明文分组进行重复处理（如图4-14所示）。

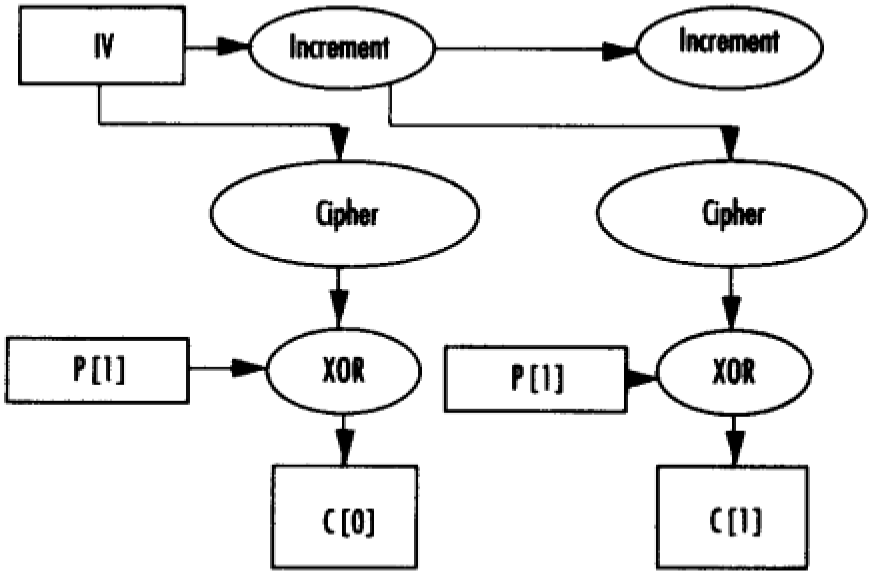


图4-14 计数器密码模式

由SP800-38A定义的“递增”函数只是一个简单的对值“1”的二进制加法。这个标准并没有指出这个字符串是big-endian还是little-endian（即如果你对链接值的字节15进行递增，那么进位将朝着字节0的方向）。

SP800-38A标准允许置换函数而不是递增函数。例如，在硬件中，一个LFSR的步进（参见第3章）会比一个加法要快，因为它没有进位。但是标准澄清说，所有的使用相同秘密密钥的链接值必须是惟一的。

1. 消息长度

在CTR情况中，我们只要把加密的输出和明文进行异或就可以。这意味着，没有任何遗留的原因能够说明为什么明文必须是密码分组长度的倍数。对于1位和万亿位的消息来讲，CTR模式的表现同样好。

CTR模式要求IV和密文一同传输——因此仍然有初始消息扩展的问题需要处理。CBC中的IV可以从密钥中获得，以同样的方式，CTR模式也可以这么做。

2. 解密

在CTR模式中，加密和解密的处理过程是相同的。这使得其实现更加简单，因为这样只需要处理很少的代码。同样地，不同于CBC模式，它只需要算法的加密方向。通过不包含ECB解密模式的支持，在实现上还能够进一步地节约空间。

3. 性能

CTR模式允许并行加密或者解密。这在软件设计中并不重要，但在硬件中则显得更加重要，在其中你可以拥有一个、两个或许多算法"核心"来实现吞吐量的线性提高。

4. 安全性

CTR模式非常依赖于链接值是惟一的这样一个条件。这是因为，如果你重用一个链接值，那么，对两个密文分组的异或就将是相应的明文分组的异或。在许多环境下，例如，当对英文ASCII文本进行加密时，这足够得到这两个分组的内容的了（你可能会认为平均32个字节的英文ASCII有41.6位的熵，这比一个单一的明文分组的长度要小）。

解决这个问题的最简单的办法，不是随机的生成IV，而是不要重用相同的密钥。这可以用相当合适的密钥生成协议（参见第5章）来实现。

5. 实现

我们再次使用那个32位快速的AES代码来构建另一种链接模式，这次是CTR链接模式。

```
ctr.c:
void AesCTRMode(unsigned char *IV,
                unsigned char *in,
                unsigned char *out,
                unsigned long  size,
                unsigned long *skey, int Nr);
{
   unsigned char buf[16];
   unsigned long x, y;
   int           z;

   while (size) {
      /* encrypt counter */
      AesEncrypt(IV, buf, skey, Nr);

      /* increment it */
      for (z = 15; z >= 0; z--) {
         if (++IV[z] & 255) break;
      }

      /* process input */
      y = (size > 16) ? 16 : size;
      for (x = 0; x < y; x++) {
         *in++ = *out++ ^ buf[x];
      }
      size -= y;
   }
}
```

在这个实现中，我们再次以字节级别来演示这种模式。第28行的异或可以更加高效地使用更大数据类型的一组异或来实现。递增（从第21行～第23行的代码）实现了一个big-endian形式的加1递增。这种递增是可行的，因为当且仅当（++*IV*[*z*]&255）的值为0时才有可能产生进位。因此，如果它不为0的话，那么就没有产生进位并且循环应该结束。

这里和CBC模式不同的是，我们没有特定的明文和密文变量。这个函数即可以用于加密也可以用于解密。我们也修改IV的值，使得该函数可以用相同的密钥被多次调用。

4.4.3 选择一个链接模式

在CBC和CTR模式之间进行选择是相当容易地。在大多数的软件实现中，它们的执行速度差不多。CTR比CBC更加灵活的地方在于它能天然地支持任意长度的明文而不需要填充或者密文窃取。从另一方面来讲，CBC是建立在现有的标准之上的。

一个比较好的简单的规则是，除非你不得不使用CBC模式，否则使用CTR。

4.5 总结

在使用一个分组密码的时候，至少应使用CTR或者CBC模式，要记住的第一件事是不要考虑认证的概念。没有哪一种模式能给你任何保证。一种常见的误解是认为CBC在这个方面是安全的。这并不正确。你之所以使用CBC或者CTR模式，是用来建立一个私有通信媒介的。

另一个比较重要的方面是大多数情况下你确实需要认证。这意味着你将同时使用一种MAC算法（参见第6章）和一种用CBC或者CTR模式的分组密码。认证对于保护你的资产以及抵抗各种随机的谕示攻击（Bernstein和Osviks攻击）是很重要的。如果不存在攻击者无时无刻对消息修改的可能，那么惟一的时间认证也算不上是一个问题了。就是说，这是一个很安全的网络，能够确保对用户的认证并且它通常也不会花费太多资源。

既然我们已经将那些重要的问题解决了，那么就可以继续研究分组密码和链接模式了。我们所要做的第一件事是给分组密码设置密钥，并且要安全地做这件事。取决于怎样使用应用程序，从哪里以及怎样得到算法的密钥是一个很大的问题。我们要做的第二件事就是给链接模式生成一个合适的IV，这一步常和密钥生成紧密相联。

1. 对分组密码设置密钥

用尽可能多的熵对分组密码设置密钥是非常重要的。如果你的密钥只有10位的熵，那么使用AES是毫无意义的。这样，穷举攻击将变得很简单而且你也不能给你的用户提供保密性。要经常使用你的应用程序所能够允许的最大的大小。AES要求你至少使用一个128位的密钥。如果您的应用程序允许性能上的不足，您最好选择一个大的密钥，这确保了在你的PRNG（或者RNG）中的任何偏差仍然能允许密钥中含有高数量的熵。

不要将密钥内嵌到你的应用程序中。这并没有任何用处。

对一个分组密码设置密钥的最常用的办法是使用一个主密钥（master key），它也叫做共享秘密。例如，一个盐化了的密码散列（salted password hash）（参见第5章）能提供一个主密钥，一个公钥加密了的随机密钥（参见第9章）能提供一个共享秘密。

一旦我们有了这个主密钥，我们就能使用一种密钥衍生算法（key derivation algorithm），

例如PKCS#5（参见第5章），衍生出一个密钥和一个IV。由于对于每个会话，其主密钥应该是随机的（即每次都处理一个新的消息），因此其衍生出来的密钥和IV也应该是随机的（如图4-15所示）。



图4-15 密钥衍生函数

不要直接用一些像口令一样简单的东西来衍生密钥（会话密钥），这是很重要的。某种形式的密钥衍生方案应该总是存在于你的加密系统流水线中。

2. 对分组密码重新设置密钥

安全地对分组密码设置密钥和对它重新设置密钥同样重要。在相当长的时间内使用相同的密钥一般来说是不安全的。从形式上来看，其限制局限于生日导论的界限之内（对于AES来说是2^{64}个分组）。但是，在那时你所能够加密的所有秘密通常超过了为支持安全地重新设置密钥所花费的代价。

根据协议是在线的还是离线的，可以用多种形式来实现密钥的重新设置。如果是在线协议，那么应该在时间和流量的基础上引发密钥的重新设置；也就是说，要在过去了一定的时间以及观察了一定数量的流量之后。所有触发密钥重新设置的信号都应该像系统中的其他消息一样经过认证。如果系统是离线的，处理就更为简单，因为只有一个成员要同意改变密钥。要根据产品和威胁向量来制定合理的方案，尤其是因为威胁并不是来自于算法的破解而是来自于通过其他手段，例如侧信道攻击或者协议的攻破来得到密钥。

一旦重新设置密钥的决定被接受（或者只是被采纳），合理地对分组密码重新设置密钥是很重要的。重新设置密钥的目标是减少因攻击者截获密钥所带来的损失。这意味着新的密钥是不能从前一个密钥中衍生出来的。但是，如果你是聪明的并且从一个主密钥中衍生出你的密钥，那么你的世界将变得更加简单。

此时，攻击者可能有少量的密钥衍生输出，但还是不够用来攻破密钥衍生函数。使用密钥衍生函数在一个给定的区间中生成一个新的密钥可以很好地解决我们的问题。

3. 双向信道

当初始化双向信道（bi-directional channel）时，一种有用的技巧是使用生成的两对密钥和IV（理想的情况是来自一个合理的密钥衍生函数）。双方都可能会生成相同的密钥对，但要以相反的顺序来使用。这具有密码学以及实际应用上的好处。

在实际应用方面，这可以允许双方能立刻发送消息并且保持IV同步。它也有助于更为简单地处理丢失的包（UDP连接的情况下）。

在密码学方面，这降低了在一个给定的对称密钥下所使用的流量。假设双方都在传输大致

相等的流量，那么攻击者现在就能够在一个给定密钥之下看到一半的密文。

4. 有损信道（Lossy Channels）

在某些通信信道上，例如UDP，包有可能会丢失，或者更糟地是，无序地到达。首先，我们来看怎样处理包数据。

在理想的协议中，尤其是当使用UDP的时候，协议应当能够处理较小数量的流量丢失，通过忽略那些丢失的数据或者请求再次传输的方法。在延迟是一个问题的情况下，通常选择UDP作为传输数据的协议。不要误认为TCP是一个安全的传输协议。即使它使用了校验和以及计数器，但数据包仍然能够被攻击者很容易地修改。所有接收到的数据都应当被怀疑。为了处理丢失，每个包都必须独立。意思是说，每个包都应当有它自己的IV。

如果你在使用CBC模式，这意味着每个包都需要一个随机IV。因为IV是随机的，所以我们也会给它附加一个计数器。该计数器对于每个包都是惟一的而且是递增的。这允许接收者能够知道它是不是以前曾经看到过的包，如果不是，那么它属于包流中的哪一个。最少情况下，这意味着每个包开销20个字节（对于AES来说），即IV用16个字节，计数器用4个字节。

如果我们使用的是CTR模式，那么每个包都需要一个惟一的IV。没有任何理由可以说明IV自身不能作为包计数器。同样地，也没有理由可以说明如果密钥是随机选择的话，IV必须要足够大。例如，如果我们知道将要发送少于2^{32}个包，就可以使用一个4字节的计数器，让剩余的计数器的低12个字节默认为0（在对包的加密过程中将递增）。这仅会增加每个包4字节的开销。要了解更多关于流协议的细节，请参考第6章。

既然所有的包都有计数器，那么就可以用它们来做些什么。我们可以丢弃旧的或者重播的包，并且可以对无序的包进行排序。有两种逻辑上的方法可用来处理无序的包。第一种方法是把计数器移动到最高的有效包（不用管那些低的计数器值的包）。这是最简单的办法，而且完全适用于许多应用程序（尽管名声不好，但UDP包很少是无序到达的）。因此，在它很少发生的事件中，容错应该是次要的。更为困难的办法是使用一个滑动窗口。这个窗口将从一个给定的基开始向上计数，当条件上升时，滑动基。例如，考虑如下的代码：

```
static unsigned long window, window_base
int isValidCtr(unsigned long ctr)
{
   /* below the window? */
   if (ctr < window_base) return 0;

   /* out of the window? */
   if (ctr - window_base > 31) { window_base = ctr; window = 0; return 1; }

   /* already seen? */
   if (window & (1UL << (ctr - window_base))) { return 0; }

   /* not seen */
   window |= 1UL << (ctr - window_base);

   /* shift window */
   while (window & 1) { window >>= 1; ++window_base; }
   return 1;
}
```

后续代码允许调用函数一次跟踪记录多达32个编号的包。它丢弃旧的和前面已经看到过的包，调节计数器，并且当可能的时候对窗口进行滑动。调用函数将从一个另外有效的函数（一个已经被认证了的包，参见第6章）给这个函数传入一个计数器，而且如果计数器有效则返回0，否则返回1。

4.5.1 荒诞的说法

这里有一些关于分组密码的比较流行的荒诞的说法。

- CBC能提供认证（或者完整性）；
- CBC只需要一个惟一的IV，而不是一个随机的；
- CTR模式是不安全的，因为明文并没有传给密码算法本身；
- 你可以使用隐藏在程序中的数据作为一个密钥；
- 修改算法（对细节进行混淆）可以使它更安全；
- 使用最大的密钥大小是“更加安全的”；
- 多次加密是“安全的”。

CBC模式并不提供认证（抱歉再次提到这个，但许多人确实对此有误解）。如果你想要进行认证（并且你经常这么做），那么对消息（或密文）使用一个MAC。CBC模式也需要不可预测的IV，而不仅仅是惟一的一个。

CTR模式证明是降低了分组密码算法的安全性的。也就是说，如果CTR模式是不安全的，那么CBC、OFB和CFB模式也一样。

把一个算法的密钥嵌入到应用程序中去是人们似乎很喜欢的一种技巧。Diebold就是这些类型中的一个，不同的是它的源代码被泄露了而且密钥也被找到了（顺便提一下，它是“F2654hD4”）。不要这么做。

对算法进行修改以实现一些个人调整是另外一种失误。它通过不允许交互来防止用户对算法的修改，那会让算法更不安全，而且最终某些人也有可能把它逆向出来。

使用最长的密钥并不总是“更安全的”或者并不是一种更好的实践。但至少从理论上来说，它是一个不错的办法。例如，如果你的RNG只有很少的偏差，那么它所生成的一个256位的字符串会比一个128位的字符串包含更多的熵（至少平均上是如此）。但在AES中，256位的密钥运行起来很慢（14轮与10轮），而且通常也需要同时运行RNG或PRNG。在可预见的未来，穷举一个128位的密钥仍然是不太可能的。

为什么更大的并不是总是更好的，关键原因是因为对于使用更大的密钥的分组密码来说，可以很容易地找到比穷举更快的攻击方法。考虑使用一个64位密钥的AES。为了破解这个算法，你不得不找到一种比2^{64}次加密更快的攻击方法。这是一个比破解完全长度的密钥更加困难的命题。算法的密钥长度只是设计者所宣称的该算法能够提供什么样的安全的一种期望。

进行多次加密，可能使用各种不同的分组密码，并不能构造一个安全的混合，它只会让设计更慢。有一种同样的逻辑“多次使用不同的密码可以让它更加安全”，它也可以说成“这些算法的特定组合会让它变得不安全”。虽然没有理由可以说明你只能相信一个论断，而不能相信其他的，但最终你只会构造出更多的私人垃圾。

再者，这也可以很容易地看出来。一个分组密码只是对一个很大的替代表的速记符号。给出任何一种特殊的替代，总可以找到另一个互补的替代，以至于当应用完一个再应用另一个时会导致一种线性变换。这种结构很容易破解。

4.5.2 提供者

有许多易读的AES加密和解密的提供者。一般来说，除非你的应用程序有非常特殊的需求，否则你（以及你的用户）不要太关心实现的细节。LibTomCrypt和OpenSSL是常见的提供者，它们都是可调用的C程序。我们可以使用前者来创建一个简单的CTR模式的例子。

建议读者阅读LibTomCrypt附带的用户手册来完全地了解这个库提供了什么。下面的代码是一个简单的程序，它对一个短的字符串使用CTR模式的AES进行加密和解密。

ctr_example.c:

```
#include <tomcrypt.h>

int main(void)
{
   symmetric_CTR ctr;
   unsigned char secretkey[16], IV[16], plaintext[32],
                 ciphertext[32], buf[32];
   int           x;

   /* setup LibTomCrypt */
   register_cipher(&aes_desc);
```

这个语句告诉LibTomCrypt用它的内部表来注册AES插件。LibTomCrypt使用一个模块化的插件系统以允许开发者能够用一种实现来替换另一种（为了增加硬件支持）。在该例中，我们正在使用内建的AES软件实现。

```
   /* somehow fill secretkey and IV ... */

```

可以很明显地看到，我们把这一块空出来了。密钥和IV可以用许多方法来衍生，其中的大部分方法我们还没有告诉你怎样使用。第5章将告诉你怎样使用PKCS#5来作为一种密钥衍生函数，第9章将告诉你怎样使用一个公钥算法来分发一个共享秘密。

```
   /* start CTR mode */
   assert(
     ctr_start(find_cipher("aes"), IV, secretkey, 16, 0,
               CTR_COUNTER_BIG_ENDIAN, &ctr) == CRYPT_OK);

```

这个函数调用使用参数对*ctr*结构进行初始化，这些参数是用于AES-128 CTR模式的，它用一个给定的IV进行加密。我们选择了一个big-endian格式的计数器以使得加密更加方便并且可以移植。这个函数有可能会失败，但有很多的办法可以用来处理错误（例如更加得体的报告），这里我们简单地使用一个断言。在实际应用中，这段代码从不会失败，但当你把像这样的代码放到更大的应用程序中去的时候，它完全有可能会失败。

```
   /* create a plaintext */
   memset(plaintext, 0, sizeof(plaintext));
```

```
        strncpy(plaintext, "hello world how are you?",
                sizeof(plaintext));

```

在把短的字符串复制到*plaintext*缓冲区中之前，我们把它完全清零。这保证我们在将要显示给用户看的最后的解密中只有0字节。

```
        /* encrypt it */
        ctr_encrypt(plaintext, ciphertext, 32, &ctr);
```

这个函数调用执行*plaintext*的AES-128 CTR模式的加密并把结果保存在*ciphertext*中。在本例中，我们加密32个字节。但要注意地是，CTR模式并不局限于处理分组大小的倍数。我们可以简单地把缓冲区变为30个字节并且它仍能发挥作用（用30来代替32）。

*ctr_encrypt*函数可以多次调用以对*plaintext*进行加密。每次传入相同的CTR结构时，它可以被修改，因此下一次调用将从前一次调用结束的地方开始。例如，

```
ctr_encrypt("hello", ciphertext, 5, &ctr);
ctr_encrypt(" world", ciphertext+5, 6, &ctr);
```

以及

```
ctr_encrypt("hello world", ciphertext, 11, &ctr);
```

执行了同样的操作。

```
        /* reset the IV */
        ctr_setiv(IV, 16, &ctr);

        /* decrypt it */
        ctr_decrypt(ciphertext, buf, 32, &ctr);
```

在用同样的CTR模式对文本进行解密之前，我们不得不重置IV。这是因为在对明文加密之后，存储在CTR结构中的链接值发生了变化。如果我们现在就开始解密，结果会不正确。

我们使用*ctr_decrypt*函数来执行从*ciphertext*到*buf*数组的解密。对于想要了解更多的读者而言，其实*ctr_decrypt*只是一个占位符，它实际上是调用*ctr_encrypt*来执行解密的。

```
        /* print it */
        for (x = 0; x < 32; x++) printf("%c", buf[x]);
        printf("\n");

        return EXIT_SUCCESS;
    }
```

此时，用户应该看到字符串“hello world how are you?”并且程序将正常地退出。

4.6 常见问题

下面的常见问题，由本书的作者所回答，它们即可以用来测试你对本章所出现的概念的理解，也可以帮助你在现实生活中实现这些概念。如果希望作者解答你的问题，请浏览www.syngress.com/solutions，然后点击“Ask the Author”表单。

问：什么是分组密码？

答：分组密码就是使用一个秘密密钥把输入（明文）变换成输出（密文）的算法。

问：分组密码的目的是什么？

答：分组密码的首先和最重要的目的是给用户提供保密性。这是通过使用秘密密钥对从明文到密文的映射进行控制来实现的。

问：分组密码算法有哪些标准？

答：FIPS 197中指定的高级加密标准（AES）。NIST的标准SP 800-38A定义了5种链接模式，包括CBC和CTR模式。

问：其他的分组密码算法呢？

答：严格地来说，NIST也把Skipjack（FIPS 185）作为是一个有效的分组密码。它比AES要慢，但由于它的大小以及使用了8位的操作，所以很适合于8位和16位的处理器。在加拿大，CSE（Communication Security Establishment，通信安全机构）正式地把CAST也作为一个标准，但除了所有NIST通过的模式之外（CSE所通过的分组密码的网站在www.cse-cst.gc.ca/services/crypto-services/crypto-algorithms-e.html）。CAST5和AES差不多快，但实现起来不如AES方便。在硬件中它的实现很大而且很困难。其他常见的对称密码如RC5、RC6、Blowfish、Twofish和Serpent都是RFC的一部分，但不是政府官方标准。欧盟的NESSIE工程选择了Anubis和Khazad作为它的128位和64位分组密码。大多数的国家都正式地把Rijndael（或者叫AES）作为他们的官方标准分组密码。

问：我可以在哪里找到像AES这样的分组密码的实现？

答：许多算法库支持各种分组密码算法。LibTomCrypt支持一个标准分组密码的组合，包括AES、Skipjack、DES、CAST5以及流行的分组密码如Blowfish、Twofish和Serpent。类似地，Crypto++支持大量的分组密码。OpenSSL也支持一些，包括AES、CAST5、DES和Blowfish。

问：什么是伪随机置换（PRP）？

答：伪随机置换就是由算法所创建的一种对符号（对于AES为整数0到$2^{128}-1$）的重新排列（因此类似于伪随机位）。一个安全的PRP的目标是，对于仅知道部分置换是不足以以一个有效的概率来确定置换的其余部分。

问：我怎样用AES来实现认证？

答：用第6章中所讲的CMAC算法。

问：CBC模式不是一个认证算法吗？

答：它可以是，但你必须要知道你在做什么。最好使用CMAC。

问：我听说CTR是不安全的，因为它不能保证认证。

答：你听到的是不对的。

问：你确定吗？

答：确定。

问：什么是IV？

答：初始向量是一个用来处理第一个分组的用于链接模式中的值。通常，前一个密文（或者计数器）用于第一个分组之后的每个分组。IV必须要和密文一起存储，而且不是秘密的。

问：我的CBC IV要是随机的，或者只是惟一的，或者是其他的什么？

答：CBC IV必须是随机的。

问：用于CTR模式的IV有什么要求？

答：CTR IV只要必须是惟一的就可以了。更精确地说，它们必须不能冲突。此意思是说在对一个消息进行加密时，其计数器的中间值必须不能等于对其他消息进行加密时的计数器的值。也就是说，这是假设你使用了相同的密钥。如果你对每个消息都改变密钥，那么你可随你所愿地重用同一个IV。

问：CTR模式相对于CBC模式有什么优点？

答：CTR模式在软件和硬件中实现都更为简单。CTR模式也可以并行地实现，这对于试图采用Gbit/s速度运行的硬件项目来说是相当重要的。CTR模式也更容易设置，因为它不需要随机的IV，这可以使得某些包算法更加有效，因为它们有更小的开销。

问：我需要一种链接模式吗？ECB模式怎么样？

答：是的，如果你要对比分组密码的分组长度（例如对于AES来说为16字节）更长的消息进行加密时，你很可能需要一种链接模式。ECB模式并不是一个实际意义上的模式。ECB意味着对几组的消息分别独立地使用密码。这完全不安全，因为这能够允许频率分析和消息确定。

问：你推荐使用哪种模式？

答：除非你想遵守某个潜在的标准，否则使用CTR模式来获得保密性，如果不是为了节约空间，那么就是为了模式的开销和执行时间效率。

问：什么是密钥衍生函数？

答：密钥衍生函数（Key Derivation Function，KDF）是一些把一个秘密映射到如密钥和IV这样的基本参数上去的函数。例如，双方可能共享一个秘密密钥而且想衍生出密钥以对他们的通信进行加密。他们可能也需要生成链接模式的IV。一个KDF能够使得他们从一个单独的共享秘密密钥中生成密钥和IV。在第5章中将对这些进行详细地介绍。

PDG

第 5 章

散列函数

本章解决方案：

- 什么是散列函数
- SHS的设计与实现
- PKCS#5密钥衍生
- 总结

☑ 总结

☑ 快速查找解决方案

☑ 常见问题

5.1 简介

安全的单向散列函数是在密码系统中像对称分组密码一样经常提及的工具。它们是高度灵活的原型，可以用来实现保密性、完整性以及认证。这一章主要处理散列函数的完整性方面。

散列函数（也正式称为伪随机函数（Pseudo Random Function，PRF）通过一个称为压缩（compression）的处理过程把任意大小的输入映射为一个固定大小的输出。这种形式的压缩并不是那种典型的数据压缩（正如你看到的.zip文件一样），而是一种不可逆的映射。非严格地讲，校验和算法是“散列函数”的一种形式，而且在许多互不相关的行业中也是这么叫它的。例如，把输入映射到散列桶（hash bucket）中是一种简单的存储任意数据的办法，它可以高效地进行查找。在密码学的意义上，散列函数必须具有两种性质才能有用：它们必须是单向的并且必须是碰撞约束（collision resistant）的。由于这些原因，简单的校验和算法和CRC对于密码学来说都不是好的散列函数。

单向则意味着给定一个散列函数的输出，推出任何关于输入的有用信息是困难的。这是一个散列算法的重要性质，因为它常用于对RNG种子数据和用户口令的处理过程中。大部分简单的校验和算法不是单向的，因为它们是线性函数。对于足够短的输入来说，由输出推导出其输入经常是一种简单的计算。

碰撞约束则意味着给定一个散列的输出，找出另外一个能产生相同输出的输入（称为碰撞）是困难的。对一个有用的散列函数，我们要求其具有两种形式的碰撞约束。预映射碰撞约束（pre-image collision resistance）（如图5-1所示）表示给定一个输出Y，找到另一个输入M’，使得M’的散列等于Y是困难的。对于数字签名，这是一个很重要的性质，因为它们仅对散列进行签名。如果这种形式的碰撞可以很容易地找到，攻击者就能够用一个签过名的消息来代替另一个消息。第二预映射碰撞约束（second pre-image collision resistance）（如图5-2所示）表示找到两个消息M1（给定的）和M2（随机选取的），使它们的散列完全符合是困难的。



图5-1 预映射碰撞约束

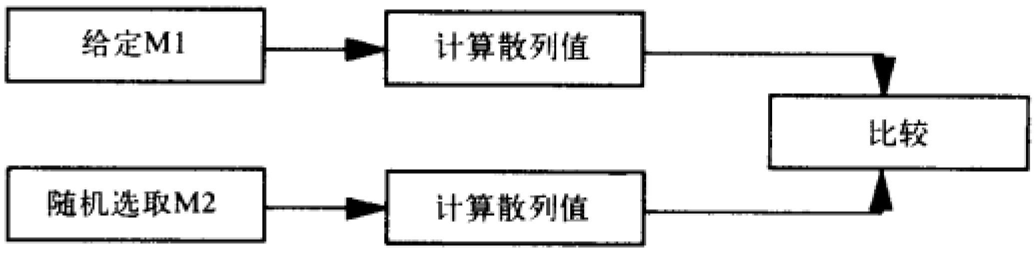


图5-2 第二预映射碰撞约束

这些年人们提出了多种安全的散列函数方案。读者也许已经听说过如MD4、MD5或者HAVAL这样的算法。所有这些算法曾经在密码学工具中占据一定的地位而且也已经被攻破了。

已经证明MD4和MD5是相当不安全的，因为它们不是抗碰撞的。HAVAL也正面临着类似的命运，但其设计者曾足够仔细地结束这个算法的设计。到目前为止，它仍然是安全的。NIST也提供了一些设计，叫做安全散列标准（Secure Hash Standard）（FIPS 180-2），它包括旧的SHA-1函数和较新的SHA-2散列家族（SHA代表安全散列算法（Secure Hash Algorithm））。在本章的其余部分，我们只介绍这些SHS算法。

SHA-1是由NIST提出的第一个散列家族。一开始，它叫做SHA，但该算法中的一个流程导致了一次调整，因此变为SHA-1（旧的标准变为SHA-0）。NIST只推荐使用SHA-1而不是SHA-0。

SHA-1是一个160位的散列函数，即其输出，也叫做摘要（digest），是160位长的。像HAVAL一样，也有针对SHA-1的简化变种的攻击，它能产生碰撞，但在写作本书的时候还没有对完整的SHA-1的攻击。目前的建议是SHA-1是不安全的，人们可以用SHA-2算法中的一种来代替它。

SHA-2是对由NIST对SHS算法的第二轮设计的一种非正式叫法。它们包含了SHA-224、SHA-256、SHA-384和SHA-512 4种算法。跟在SHA后面的数字表明其摘要长度。在SHA-2系列中，实际上只有两种算法。SHA-224使用的是SHA-256的算法，只不过做了一些较小的修改并把输出截短到224位。类似地，SHA-384使用的是SHA-512并截短了输出。目前的建议是至少使用SHA-256作为默认的散列算法，尤其是当你使用AES-128来实现保密性的话（我们将在稍后看到）。

5.1.1 散列摘要长度

你也许想知道所有这些大小的散列摘要都是从哪里来的。为什么SHA-2从256位开始并且一直增加到512（在最初的发布版本之后，SHA-224被加到了SHS规范中）？

可以证明，对一个散列的碰撞约束并不像人们所希望的那种线性。例如，在SHA-256中一

个第二预映射碰撞的概率并不是人们认为的$1/2^{256}$，而仅仅至少是$1/2^{128}$。一个叫做生日悖论（birthday paradox）的研究表明（粗略地），在一个房间里的23个人具有同一天生日的概率大概为50%。

这是因为有23C2 = 253（读做“23选择2”）个惟一的对。每一对都有364/365的生日不相同的机率。所有的对都不相同的机率是由这个分数的253次方所给出的。注意，一个事件发生的概率和它相反的情况的概率之和必须为1，我们取最终的结果并用1减去它，这样就可以得到一个很好的针对所有生日是符合的情况概率估计。可以证明它只有50%多一点点。

当数值n增长时，nC2运算非常近似等于n^2，因此对于2^{128}个散列我们有2^{256}个对并且期望能找到一个碰撞。

实际上，散列算法在强度上只有它们摘要大小的一半。SHA-256可以在2^{128}范围内找到碰撞，SHA-512在2^{256}范围内，等等。一个重要的设计原则是，确保你的所有原型具有相同的“位长”。把AES-256和SHA-1一起使用没有任何意义（至少是直接使用），因为这个散列只输出160位。生日悖论在这个问题中并没有多少作用，但是它们确实会影响数字签名，我们稍后将看到。

SHA-1的输出大小是160位实际上是因为经常使用RSA-1024（参见第9章）。攻破一个1024位的RSA大概需要2^{86}的工作，这可以很好地和找到一个散列碰撞的2^{80}的困难性相比较。这意味着，攻击者会花费足够的时间以试图找出另外一个文档使得它和目标文档相碰撞，然后攻破RSA密钥自身。

人们应当避免陷入到长度不相符的情况中。与SHA-256一起使用RSA-1024并不是一种糟糕的方法，但你应该清楚地意识到这个组合的强度只是86位而不是128位。类似地，与SHA-1一起使用RSA-2048（112位的强度），攻击者只要找到一个碰撞而不用攻破RSA密钥（这是更加困难的）就可以。

表5-1表示了对于所要求的给定位强度应该使用哪些标准。注意到表示使用哪个SHS的那一列只是最小的建议。如果你的目标只是112位的安全性，可以安全地使用SHA-256。更需要注意地是并不是通过使用一个更长的散列函数来得到安全强度的。例如，你正在使用ECC-192并选择了SHA-512，你仍然只能有96位的安全性（保证其他的方面都是安全的）。明智地选择你的原型。

表5-1 位强度和散列标准组合

位强度	ECC强度	RSA强度	可以使用的SHS
80	ECC-192*	RSA-1024	SHA-1
112	ECC-224	RSA-2048	SHA-224
128	ECC-256		SHA-256
192	ECC-384		SHA-384
256	ECC-521		SHA-512

*从技术上来说，ECC-192需要2^{96}的工作来破解，但这只是NIST提供的最小的标准ECC曲线。

许多（聪明的）人写了一些关于选择什么样的密钥大小才能最大限度地保证安全的文章。我们仍然把它简单地介绍给你。如果你的应用程序能够允许的话把安全目标至少定为128位，最好更多。一般情况下，更大的密钥意味着更慢的算法，因此考虑时间的限制是很重要的。更

小的密钥则意味着，你在请求某人把许多事情放在一起，然后破解你的密码算法。最后，如果过分地担心你所使用的密钥大小而不关心应用程度的整体效果，那么作为一个加密人员，你没有做一项好的工作。

来自安全研究人员的忠告

对MD5散列的MD5CRK攻击

在一个固定的函数中找到一个碰撞的常用方法，如果实际上不是存储了一个巨大的值的列表并且进行比较的话，那么就是循环查找（cycle finding）。这种攻击是通过在它的输出上进行函数的迭代来工作的。以两个或者更多个不同的初始值开始，并且一直循环直到它们中的两个发生碰撞为止。例如，如果用户A以A[-1]开始，用户B以B[-1]开始，并且A[-1]不等于B[-1]，那么我们计算

A[i] = Hash（A[i-1]）

B[i] = Hash（B[i-1]）

直到A[i]等于B[i]时停止。显然，如果你想进行分布式攻击来进行在线比较是很麻烦的。但是，把A[i]和B[i]的整个列表存储起来用来比较也是非常低效的。一种聪明的优化办法是只存储特异点（distinguished point）。通常，它们是通过一种特殊的位类型来进行区分的。例如，只存储前l位为0的散列值。

现在，如果它们能够碰撞，那么它们也能够产生会碰撞的特异点。l的值在收集所占用的内存方面以及效率之间提供了一种平衡。位越多，表就越小，但这也会占用用户更长的时间来报告特异点。所用的位越少，表就越大，而且查找的速度就越慢。

5.2 SHS的设计与实现

先前提到，SHS FIPS 180-2由3种不同的算法所组成：SHA-1、SHA-256和SHA-512。由最后两个可以构造出变形的SHA-224和SHA-384算法。我们将首先考虑这3种不同的算法。

所有的这3种算法都遵循相同的基本设计流程。即提取一个消息分组，将其扩展，然后把它传给一个会修改内部状态（它是消息摘要的大小）的压缩函数。所有的这3种算法都使用了一种对消息进行填充的技术，叫做MD强化（MD strengthening）。

在图5-3中我们能够看到两个消息分组M[0,1]的散列值计算的流程。M[0]首先被扩展，然后用现有的散列状态进行压缩。其输出是新的散列状态。接着，我们使用消息摘要（MD）强化技术对M[1]进行填充，扩展它并且用散列状态进行压缩。因为它是最后一个分组，所以压缩函数的输出就是散列摘要。

用分组密码方面的术语来描述这3种散列是相当简单的。消息分组（密钥）被扩展为一个轮密钥的集合。然后使用这些轮密钥来对当前的散列状态进行加密，这是在执行把消息压缩到更小的散列状态的操作。这种结构也可以把一个普通的分组密码转化为一个散列函数。例如，对于AES，我们有

S[i] := S[i-1] xor AES（M[i],S[i-1]）

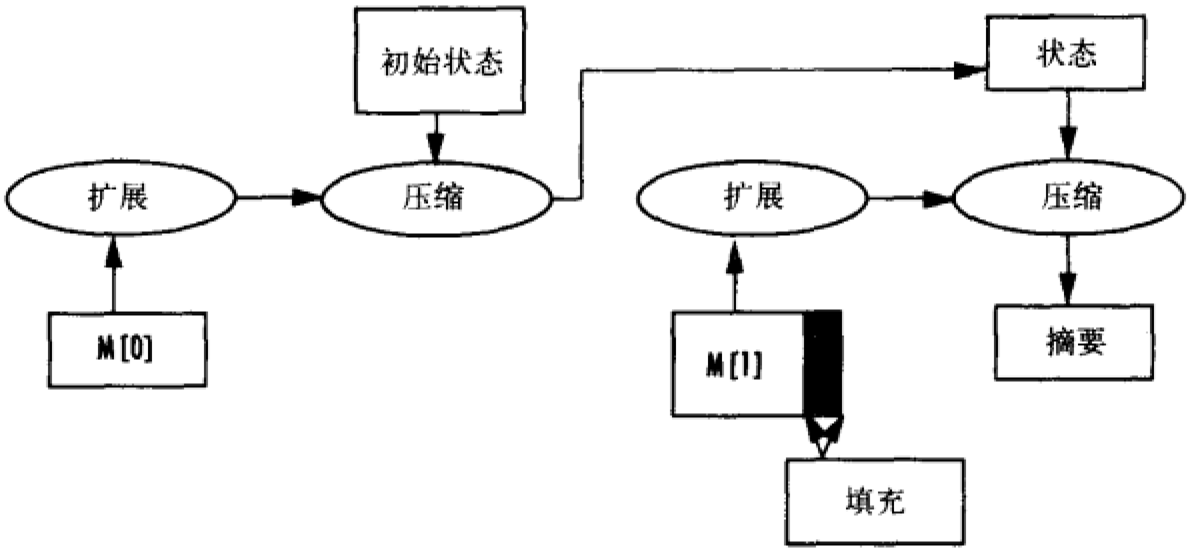


图5-3 一个两分组消息的散列

其中AES（M[i]、S[i-1]）使用密钥M[i]对前一个状态S[i-1]加密。我们使用一个固定的已知的值S[-1]，并通过使用MD5强化对消息进行填充就能构造出了一个安全的散列算法。使用传统的密码算法的问题是密钥和密文输出太小。例如，使用AES-256，每次调用能够压缩32个字节并产生一个128位的摘要。另一方面，SHA-1每次调用能够压缩64个字节并且产生一个160位的摘要。

5.2.1 MD 强化

MD强化处理过程起初是由Rivest博士为MD系列的散列算法所发明的。其目标是通过对长度进行编码后，作为消息的一部分来阻止一种前缀（prefix）和后缀（suffix）的组合攻击。

填充过程如下所示。

1. 给消息添加一个单独的位，其值为1。
2. 添加足够的0位使得消息的长度和$w-1$模w同余。
3. 以big-endian格式把消息的长度作为一个l位的整数进行添加。

其中w是散列函数的分组长度，l是能够编码的最大消息长度的位数。对于SHA-1、SHA-224和SHA-256，w=512且l=64。对于SHA-384和SHA-512，w=1024且l=128。

5.2.2 SHA-1的设计

SHA-1是MD系列之后的目前为止最为常用的散列函数。在刚发明它的时候，它是散列算法中惟一一个能够产生160位的摘要并且仍然保持相当水平的效率的散列函数。我们将把SHA-1设计分成3个组件：状态、扩展函数和压缩函数。

1. SHA-1的状态

SHA-1的状态是一个包含5个32位字的数组，用S[0...4]来表示，它存储了如下的5个初始值：

```
S[0] = 0x67452301;
S[1] = 0xefcdab89;
S[2] = 0x98badcfe;
S[3] = 0x10325476;
S[4] = 0xc3d2e1f0;
```

其压缩函数的每次调用都修改该SHA-1的状态。SHA-1状态的最终值就是散列摘要。

2. SHA-1的扩展

SHA-1以64个字节的分组来处理消息。不管我们正在扩展哪个分组（第一个或者最后一个需要填充），其扩展处理都是一样的。扩展函数使用了一个包含80个32位字的数组，用W[0...79]来表示。前16个字是从消息分组中以big-endian格式加载的。

接着的64个字是使用下面的代码生成的。

```
for (x = 16; x < 80; x++) {
   W[i] = ROL(W[i-3] ^ W[i-8] ^ W[i-14] ^ W[i-16], 1);
}
```

其中ROL（x, 1）是一个循环左移1位的操作。此时，我们已经完全把64个字节的消息分组扩展到了压缩函数所需要的轮密钥中。想要了解更多内容的读者也许想知道SHA-0的定义中有没有这个移位操作。如果没有这个移位操作，那么散列函数将不会具有用来抵抗碰撞搜索而要求的80位的强度。

3. SHA-1的压缩

其压缩函数是一种类型的80轮的Feistel网络（如果你还不知道它，也不要担心）。在每一轮中，取自状态的3个字被传入到一个轮函数中并且和剩余的两个字进行混合。SHA-1使用4种类型的轮函数，每一种迭代20次，总共80轮。

散列状态首先必须从S[ ]数组中复制到某个局部变量中。我们可以称之为{a, b, c, d, e}，其中a=S[0]，b=S[1]，如此等等。其轮结构类似于下面这样。

```
FFx(a,b,c,d,e,x) \
   e = (ROL(a, 5) + Fx(b,c,d) + e + W[x] + Kx); b = ROL(b, 30);
```

其中*Fx*(*x*, *y*, *z*)是用于第*x*个组的轮函数。在每一轮之后，状态中的字要进行交换，使得*e*变成*d*，*d*变成*c*，等等。*Kx*表示第*x*轮的轮常量。每个轮组都有它自己的常量。4个轮函数以下面的代码给出。

```
#define F0(x,y,z)   (z ^ (x & (y ^ z)))
#define F1(x,y,z)   (x ^ y ^ z)
#define F2(x,y,z)   ((x & y) | (z & (x | y)))
#define F3(x,y,z)   (x ^ y ^ z)
```

轮常量如下。

$K_{0...19}$ = 0x5a827999
$K_{20...39}$ = 0x6ed9eba1
$K_{40...59}$ = 0x8f1bbcdc
$K_{60...79}$ = 0xca62c1d6

在每种轮的20轮结束之后，5个字{a, b, c, d, e}会和散列状态中的相应字进行相加（看成是整数，并且要模2^{32}）。轮函数的另外一种视图如图5-4所示。

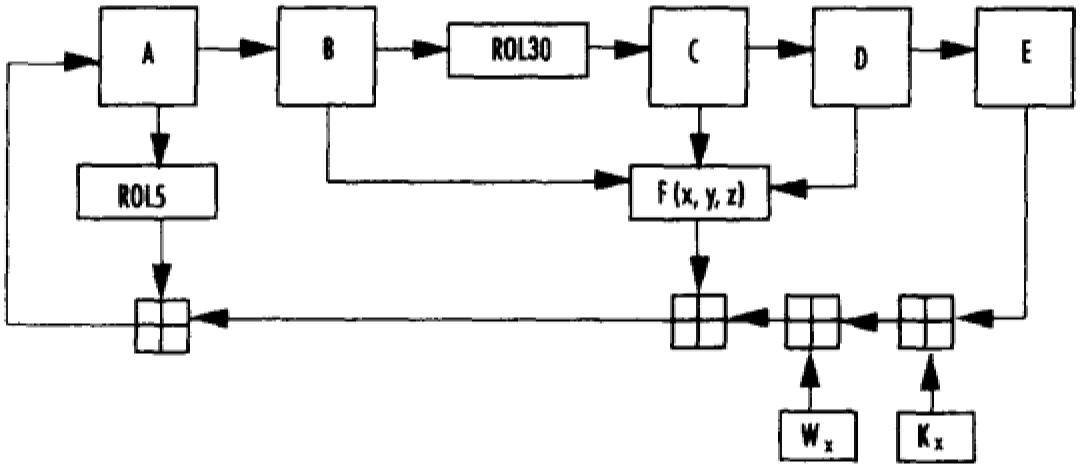


图5-4 SHA-1轮函数

4. SHA-1的实现

我们的SHA-1实现是对标准的一种直接翻译，而且没有使用一组常用的优化，这些优化我们将在代码中提到。

sha1.c:

```
#if defined(__x86_64__)
   typedef unsigned ulong32;
#else
   typedef unsigned long ulong32;
#endif

/* Helpful macros */
#define STORE32H(x, y)                                    \
     { (y)[0] = (unsigned char)(((x)>>24)&255);   \
       (y)[1] = (unsigned char)(((x)>>16)&255);   \
       (y)[2] = (unsigned char)(((x)>>8)&255);    \
       (y)[3] = (unsigned char)((x)&255); }

#define LOAD32H(x, y)                                    \
     { x = ((ulong32)((y)[0] & 255)<<24) | \
           ((ulong32)((y)[1] & 255)<<16) | \
           ((ulong32)((y)[2] & 255)<<8)  | \
           ((ulong32)((y)[3] & 255)); }

#define ROL(x, y) \
((((ulong32)(x)<<(ulong32)((y)&31)) |   \
(((ulong32)(x)&0xFFFFFFFFUL)>> \
(ulong32)(32-((y)&31)))) & 0xFFFFFFFFUL)
```

在这个实现中我们所熟悉的宏又出现了。这些宏在可移植的代码中是一个“救命者”，因为它们消除了我们可能会碰到的任何endian问题。

```
#define F0(x,y,z)  (z ^ (x & (y ^ z)))
#define F1(x,y,z)  (x ^ y ^ z)
#define F2(x,y,z)  ((x & y) | (z & (x | y)))
#define F3(x,y,z)  (x ^ y ^ z)
```

这些是SHA-1的轮函数。

```
typedef struct {
   unsigned char buf[64];
   unsigned long buflen, msglen;
```

```
    ulong32        S[5];
} sha1_state;
```

这个结构存储了一个SHA-1状态。这允许我们能够用对sha1_process的多次调用来处理消息。例如，如果我们正在处理一个消息，它是一个数据流或者太大了以致于不能一次全部放到内存中，那么我们就可以使用多次调用这个处理函数来处理整个消息。

```
void sha1_init(sha1_state *md)
{
    md->S[0] = 0x67452301;
    md->S[1] = 0xefcdab89;
    md->S[2] = 0x98badcfe;
    md->S[3] = 0x10325476;
    md->S[4] = 0xc3d2e1f0;
    md->buflen = md->msglen = 0;
}
```

这个函数把SHA-1状态初始为默认状态。我们把S[ ]数组设置为SHA-1所默认的，并且把缓冲区和消息长度设为0。缓冲区长度（buflen）变量计算在当前的分组中有多少个字节。当它达到64时，就要调用压缩函数来对数据进行压缩。消息长度（msglen）变量计算整个消息的长度。在我们的例子中，是计算字节，该意思是说，这个程序仅限用于$2^{32}-1$个字节的消息。

```
static void sha1_compress(sha1_state *md)
{
   ulong32 W[80], a, b, c, d, e, t;
   unsigned x;

   /* load W[0..15] */
   for (x = 0; x < 16; x++) {
       LOAD32H(W[x], md->buf + 4 * x);
   }
```

这个函数把消息分组进行扩展并压缩到状态中。第一个循环以big-endian格式，利用LOAD32H宏把64个字节的分组加载到W[0...15]中。

```
   /* compute W[16...79] */
   for (x = 16; x < 80; x++) {
       W[x] = ROL(W[x-3] ^ W[x-8] ^ W[x-14] ^ W[x-16], 1);
   }
```

这个循环产生W[ ]数组中其余的条目。类似于AES密钥调度（参见第4章），它是一种形式的移位寄存器。

提示 *类似于AES密钥调度，我们可以在有限的内存环境中对SHA-1进行优化。通过运行扩展函数，我们只需要16个32位字（64个字节），而且不需要全部的80个32位字(320个字节)。这种优化是利用了这样一个事实，即W[x]和W[x-16]可以在内存中重叠。*

```
   /* load a copy of the state */
   a = md->S[0]; b = md->S[1]; c = md->S[2];
   d = md->S[3]; e = md->S[4];

   /* 20 rounds */
   for (x = 0; x < 20; x++) {
      e = (ROL(a, 5) + F0(b,c,d) + e + W[x] + 0x5a827999);
```

```
068         b = ROL(b, 30);
069         t = e; e = d; d = c; c = b; b = a; a = t;
070      }
```

这个循环实现了SHA-1压缩函数的前20轮。这里我们把循环组和起来以节省空间。也有另外两种常用的方法来实现它。一种是把它部分分开，即复制循环体5次。这使得我们可以使用寄存器重命名来代替交换（第69行代码）。通过使用如下的一个宏。

```
#define FF0(a,b,c,d,e,i) \
e = (ROLc(a, 5) + F0(b,c,d) + e + W[i] + 0x5a827999UL); \
b = ROLc(b, 30);
```

这个循环可以如下进行实现。

```
for (x = 0; x < 20; ) {
  FF0(a,b,c,d,e,x++);
  FF0(e,a,b,c,d,x++);
  FF0(d,e,a,b,c,x++);
  FF0(c,d,e,a,b,x++);
  FF0(b,c,d,e,a,x++);
}
```

这省去了所有的交换操作而且执行起来更加高效（虽然只有5倍大）。另一种优化的方法是把整个循环都分开来，并用轮数来代替“x++”。这种拆分在现代处理器（带有好的分支预测）上很少能产生出好处而且是污染指令缓存的主要因素。

```
072      /* 20 rounds */
073      for (; x < 40; x++) {
074         e = (ROL(a, 5) + F1(b,c,d) + e + W[x] + 0x6ed9eba1);
075         b = ROL(b, 30);
076         t = e; e = d; d = c; c = b; b = a; a = t;
077      }
078
079      /* 20 rounds */
080      for (; x < 60; x++) {
081         e = (ROL(a, 5) + F2(b,c,d) + e + W[x] + 0x8f1bbcdc);
082         b = ROL(b, 30);
083         t = e; e = d; d = c; c = b; b = a; a = t;
084      }
085
086      /* 20 rounds */
087      for (; x < 80; x++) {
088         e = (ROL(a, 5) + F3(b,c,d) + e + W[x] + 0xca62c1d6);
089         b = ROL(b, 30);
090         t = e; e = d; d = c; c = b; b = a; a = t;
091      }
```

这3个循环实现了SHA-1压缩的最后60轮。我们可以为了额外的性能提升而像前一轮一样使用相同的代码风格把它们拆分开来。

```
093      /* update state */
094      md->S[0] += a;
095      md->S[1] += b;
096      md->S[2] += c;
097      md->S[3] += d;
098      md->S[4] += e;
099   }
```

最后一点是，代码通过把状态的副本加到它自身上去来修改状态的。这个加法应该是模2^{32}的。我们不需要对结果进行约简，因为移位宏是显示地在32位的范围内进行掩位操作的。

```
void sha1_process(         sha1_state *md,
                  const unsigned char *buf,
                        unsigned long  len)
{
```

这个函数是调用函数用来把消息字节加到散列中去的。这个函数把消息字节从buf中复制到内部缓冲区并且当积累到64个字节时，对它们进行压缩。

```
   unsigned long x, y;

   while (len) {
       x     = (64 - md->buflen) < len ? 64 - md->buflen : len;
       len -= x;
```

我们的缓冲区只有64个字节的长度，因此必须防止复制过多的字节给它。在这之后，*x*包含了我们需要复制的字节的数目。

```
       /* copy x bytes from buf to the buffer */
       for (y = 0; y < x; y++) {
           md->buf[md->buflen++] = *buf++;
       }

       if (md->buflen == 64) {
          sha1_compress(md);
          md->buflen  = 0;
          md->msglen += 64;
       }
   }
}
```

在复制完字节之后，要检查缓冲区是否有64个字节，如果有，我们就调用压缩函数。压缩完之后，重置缓冲区长度并修改消息长度。我们可以对这个处理函数进行一种优化，它是一种执行零复制压缩（zero-copy compression）的方法。

假设当进入这个函数时，buflen值为0且你想要再多散列63个字节的数据。为什么一定要把数据在散列压缩之前复制到buf[ ]数组中去呢？通过直接对用户提供的缓冲区进行散列，我们就可以省去开销很大的双缓冲。这种优化还可以进一步使用。假设buflen不为0，但我们要对大于64个字节的数据进行散列。我们可以对缓冲区加倍直到填充buflen的时候，那时，如果剩余足够的字节，那么我们就可以以零复制足够多的64个字节的分组。

这种技巧是非常容易实现的，而且可以使性能提高几个百分点。

```
void sha1_done(   sha1_state *md,
               unsigned char *dst)
{
```

这个函数终止散列并且把摘要输出到dst缓冲区中。

```
   ulong32 l1, l2, i;

   /* compute final length as 8*md->msglen */
   md->msglen += md->buflen;
```

```
131       l2 = md->msglen >> 29;
132       l1 = (md->msglen << 3) & 0xFFFFFFFF;
```

从技术上看，SHA-1支持高达$2^{64}-1$位的消息，但是，由于实际原因，我们把自己限制为$2^{32}-1$字节。当我们能对长度进行编码之前，必须把它作为位数进行编码。我们提取出消息长度的高位（第131行代码）并左移3位，模拟一个乘以8的运算。此时，64位的值$12*2^{32}+11$代表了用位表示的填充机制所需要的消息长度。

```
134       /* add the padding bit */
135       md->buf[md->buflen++] = 0x80;
```

这是起始填充位。因为我们是以字节单元来处理的，所以总是以一个字节的界限来对齐。SHA-1是big-endian格式的，因此值为1的位会变化成0x80（带有7个0填充位）。

```
137       /* if the current len > 56 we have to finish this block */
138       if (md->buflen > 56) {
139          while (md->buflen < 64) {
140              md->buf[md->buflen++] = 0x00;
141          }
142          sha1_compress(md);
143          md->buflen = 0;
144       }
```

如果当前的分组足够大，能容纳64位的长度，那么我们必须用0字节来进行填充以达到64个字节。然后我们进行压缩并重置缓冲区长度计数器。

```
146       /* now pad until we are at pos 56 */
147       while (md->buflen < 56) {
148           md->buf[md->buflen++] = 0x00;
149       }
```

此时，可以保证buflen<56。我们用足够的0字节来填充直到我们达到第56个位置。

```
151       /* store the length */
152       STORE32H(l2, md->buf + 56);
153       STORE32H(l1, md->buf + 60);
```

我们把消息的长度看成是一个64位的big-endian格式的数字来存储。我们通过执行两个big-endian的32位存储来模仿这个操作。

```
155       /* compress */
156       sha1_compress(md);
```

一个包含填充的最终的压缩结束这个散列。

```
158       /* extract the state */
159       for (i = 0; i < 5; i++) {
160          STORE32H(md->S[i], dst + i*4);
161       }
162    }
```

现在我们提取其摘要，它是SHA-1状态的5个32位字。此时，dst[0...19]包含了被散列的消息的消息摘要。

```
164    void sha1_memory(const unsigned char *in,
165                           unsigned long  len,
166                           unsigned char *dst)
```

```
{
   sha1_state md;
   sha1_init(&md);
   sha1_process(&md, in, len);
   sha1_done(&md, dst);
}
```

这个函数是一个很好的辅助函数。它对一个完全的消息进行散列，并且在一个单独的函数调用中输出摘要。这对于能够在内存中放得下的小的缓冲区是很有用的——例如，用于散列口令（passwords）和盐渍（salts）。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(void)
{
  static const struct {
      char *msg;
      unsigned char hash[20];
  } tests[] = {
    { "abc",
      { 0xa9, 0x99, 0x3e, 0x36, 0x47, 0x06, 0x81, 0x6a,
        0xba, 0x3e, 0x25, 0x71, 0x78, 0x50, 0xc2, 0x6c,
        0x9c, 0xd0, 0xd8, 0x9d }
    },
    { "abcdbcdecdefdefgefghfghighi"
      "jhijkijkljklmklmnlmnomnopnopq",
      { 0x84, 0x98, 0x3E, 0x44, 0x1C, 0x3B, 0xD2, 0x6E,
        0xBA, 0xAE, 0x4A, 0xA1, 0xF9, 0x51, 0x29, 0xE5,
        0xE5, 0x46, 0x70, 0xF1 }
    }
  };
  int i;
  unsigned char tmp[20];

  for (i = 0; i < 2; i++) {
     sha1_memory((unsigned char *)tests[i].msg,
                 strlen(tests[i].msg), tmp);
     if (memcmp(tests[i].hash, tmp, 20)) {
         printf("Failed test %d\n", i);
         return EXIT_FAILURE;
     }
  }
  printf("SHA-1 Passed\n");
  return EXIT_SUCCESS;
}
```

我们的演示程序使用了两个标准的SHA-1测试向量来确定我们的代码是不是真的有效。在测试组合中，我们遗漏了几个极端情形（它们并不是FIPS的标准），它们是：

1. 一个长度为0的消息（null）。

其散列为：da39a3ee5e6b4b0d3255bfef95601890afd80709

2. 一个精确的64个字节长的消息。

64个0字节的散列：c8d7d0ef0eedfa82d2ea1aa592845b9a6d4b02b7

3. 一个精确的56个字节长的消息。

56个0字节的散列：9438e360f578e12c0e0e8ed28e2c125c1cefee16

4. 一个是64个字节的某个倍数长的消息

128个0字节的散列：0ae4f711ef5d6e9d26c611fd2c8c8ac45ecbf9e7

FIPS 800-2标准提供了3个测试向量，在我们例子中使用了两个。第三个是许多的a组成的字符串，它的散列结果应该为34aa973c d4c4daa4 f61eeb2b dbad2731 6534016f。

通过对其他的NIST标准的观察我们可以发现，其测试向量不完整的情况并不是偶然的。顺便提一下，这么长的时间以来NIST验证的证书花费了多少？

5.2.3 SHA-256的设计

SHA-256的设计和SHA-1的类似。它使用一个256位的状态，分为8个32位的字，用S[0...7]来表示。虽然它和SHA-1具有同样的扩展和压缩的感觉，但实际操作更加复杂。SHA-256使用一个独立的重复64次的轮函数。

SHA-256使用8个如下定义的简单函数的集合。

```
#define Ch(x,y,z)          (z ^ (x & (y ^ z)))
#define Maj(x,y,z)         (((x | y) & z) | (x & y))
```

这些是用于SHA-256轮函数中的非线性函数。它们提供了设计所要求的线性复杂度，以确保该函数是单向的并且其差别是很难控制的。没有这些函数，SHA-256将会是完全线性的（除了我们将要看到的整数加法之外），而且碰撞会很容易地找到。

```
#define S(x, n)            ROR((x),(n))
#define R(x, n)            (((x)&0xFFFFFFFFUL)>>(n))
```

这些分别是循环右移和逻辑右移宏。对于想要了解更多内容的读者可能会注意到，S()宏执行了一个循环操作而R()宏执行了一个移动操作。这些宏是基于SHA-256的早期设计。我们仍然把它保留下来的部分原因是因为它很有趣，另一部分原因是因为它是LibTomCrypt所使用的。

```
#define Sigma0(x)          (S(x, 2) ^ S(x, 13) ^ S(x, 22))
#define Sigma1(x)          (S(x, 6) ^ S(x, 11) ^ S(x, 25))
```

这两个函数用在轮函数里以促进扩散（diffusion）。它们通过用于SHA-1中的5和30位来代替循环。现在，输入中的1位的差别将传播到输出中的其他两个位。通过把它只放在右边，助于通过这种设计来促进快速的扩散。如果我们研究SHA-256的分组流程图（如图5-5所示），我们就能够看到Sigma0和Sigma1的输出实际上又传回了来自输入的状态中的那个字。此意思是说在一轮之后，一个独立的位会影响其周围字的3个位。在两轮之后，它至少影响了9位，等等。这种反馈机制到目前为止已被证明在抵抗密码分析方面是非常有效的。

```
#define Gamma0(x)          (S(x, 7) ^ S(x, 18) ^ R(x, 3))
#define Gamma1(x)          (S(x, 17) ^ S(x, 19) ^ R(x, 10))
```

这两个函数用于扩展阶段以促进碰撞约束。在SHA-256（以及FIPS 180-2）标准中，它们使用大写和小写希腊sigma来表示Sigma和Gamma函数。我们把小写的sigma函数重命名为Gamma以防止在源代码中引起混淆。

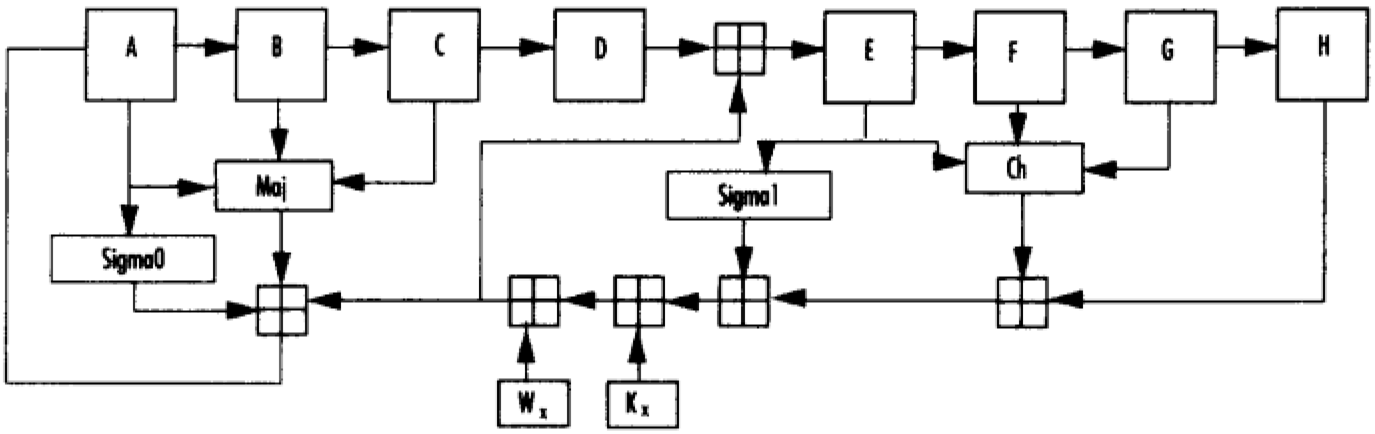


图5-5 SHA-256压缩设计

1. SHA-256的状态

SHA-256的初始状态是8个32位字，用S[0...7]来表示，如下定义。

```
S[0] = 0x6A09E667;
S[1] = 0xBB67AE85;
S[2] = 0x3C6EF372;
S[3] = 0xA54FF53A;
S[4] = 0x510E527F;
S[5] = 0x9B05688C;
S[6] = 0x1F83D9AB;
S[7] = 0x5BE0CD19;
```

2. SHA-256的扩展

类似于SHA-1，消息分组在使用之前要进行扩展。消息分组被扩展成64个32位字。前16个字是从消息分组中的64个字节以big-endian的格式加载而来的。接下来的48个字是使用如下的循环来计算得到的。

```
for (i = 16; i < 64; i++) {
   W[i] = Gamma1(W[i - 2]) + W[i - 7] +
          Gamma0(W[i - 15]) + W[i - 16];
}
```

3. SHA-256的压缩

压缩以把SHA-256的状态从S[0...7]复制到{a, b, c, d, e, f, g, h}中开始。接着执行64个不同的轮。第x轮的结构如下所示。

```
t0 = h + Sigma1(e) + Ch(e, f, g) + K[x] + W[x];
t1 = Sigma0(a) + Maj(a, b, c);
d += t0;
h  = t0 + t1;
```

在每一轮之后，这些字将进行循环，使得h变成g，g变成f，f变成e，等等，K数组是一个固定的包含64个32位字的数组（参照实现一节）。在第64轮结束之后，字{a, b, c, d, e, f, g, h}分别和散列状态进行相加。在压缩完最后一个消息分组之后的散列状态就是消息摘要。

把这个流程图和SHA-1的相比较（如图5.4），我们能够看到它们使用了两个并行的非线性的函数，两个更加复杂的扩散原型（Sigma0和Sigma1）以及更多的反馈（到字A和E）。这些改变使得新的SHS算法具有更高的扩展能力，而且也不太可能受目前困扰着MD5和SHA-1算法的同样类型攻击的影响。

4. SHA-256的实现

我们的SHA-256的实现直接采用了SHA-1的实现框架。在某种程度上，这也突出了这两个散列算法之间的相似性。这突出表明了这样一个事实，即我们不想写两个不同的散列接口，一方面是懒的原因，另一方面是这也会让代码更加难用。

```
#if defined(__x86_64__)
   typedef unsigned ulong32;
#else
   typedef unsigned long ulong32;
#endif

/* Helpful macros */
#define STORE32H(x, y)                                  \
     { (y)[0] = (unsigned char)(((x)>>24)&255);    \
       (y)[1] = (unsigned char)(((x)>>16)&255);    \
       (y)[2] = (unsigned char)(((x)>>8)&255);     \
       (y)[3] = (unsigned char)((x)&255); }

#define LOAD32H(x, y)                                  \
     { x = ((ulong32)((y)[0] & 255)<<24) | \
           ((ulong32)((y)[1] & 255)<<16) | \
           ((ulong32)((y)[2] & 255)<<8)  | \
           ((ulong32)((y)[3] & 255)); }

#define ROR(x, y) \
((((ulong32)(x)>>(ulong32)((y)&31)) |  \
(((ulong32)(x)&0xFFFFFFFFUL)<< \
(ulong32)(32-((y)&31)))) & 0xFFFFFFFFUL)
```

到目前为止，还是非常类似于SHA-1的，不同的是我们使用的是一个循环右移，而不是左移。

```
#define Ch(x,y,z)        (z ^ (x & (y ^ z)))
#define Maj(x,y,z)       (((x | y) & z) | (x & y))
#define S(x, n)          ROR((x),(n))
#define R(x, n)          (((x)&0xFFFFFFFFUL)>>(n))
#define Sigma0(x)        (S(x, 2) ^ S(x, 13) ^ S(x, 22))
#define Sigma1(x)        (S(x, 6) ^ S(x, 11) ^ S(x, 25))
#define Gamma0(x)        (S(x, 7) ^ S(x, 18) ^ R(x, 3))
#define Gamma1(x)        (S(x, 17) ^ S(x, 19) ^ R(x, 10))
```

这些是用于扩展以及压缩阶段的SHA-256的宏。

```
typedef struct {
   unsigned char buf[64];
   unsigned long buflen, msglen;
   ulong32       S[8];
} sha256_state;
```

含有更长的S[ ]数组的SHA-256的状态。其他的变量和SHA-1代码中的目的一样。

```
static const ulong32 K[64] = {
    0x428a2f98UL, 0x71374491UL, 0xb5c0fbcfUL, 0xe9b5dba5UL,
    0x3956c25bUL, 0x59f111f1UL, 0x923f82a4UL, 0xab1c5ed5UL,
    0xd807aa98UL, 0x12835b01UL, 0x243185beUL, 0x550c7dc3UL,
    0x72be5d74UL, 0x80deb1feUL, 0x9bdc06a7UL, 0xc19bf174UL,
    0xe49b69c1UL, 0xefbe4786UL, 0x0fc19dc6UL, 0x240ca1ccUL,
```

```
    0x2de92c6fUL, 0x4a7484aaUL, 0x5cb0a9dcUL, 0x76f988daUL,
    0x983e5152UL, 0xa831c66dUL, 0xb00327c8UL, 0xbf597fc7UL,
    0xc6e00bf3UL, 0xd5a79147UL, 0x06ca6351UL, 0x14292967UL,
    0x27b70a85UL, 0x2e1b2138UL, 0x4d2c6dfcUL, 0x53380d13UL,
    0x650a7354UL, 0x766a0abbUL, 0x81c2c92eUL, 0x92722c85UL,
    0xa2bfe8a1UL, 0xa81a664bUL, 0xc24b8b70UL, 0xc76c51a3UL,
    0xd192e819UL, 0xd6990624UL, 0xf40e3585UL, 0x106aa070UL,
    0x19a4c116UL, 0x1e376c08UL, 0x2748774cUL, 0x34b0bcb5UL,
    0x391c0cb3UL, 0x4ed8aa4aUL, 0x5b9cca4fUL, 0x682e6ff3UL,
    0x748f82eeUL, 0x78a5636fUL, 0x84c87814UL, 0x8cc70208UL,
    0x90befffaUL, 0xa4506cebUL, 0xbef9a3f7UL, 0xc67178f2UL
};
```

这是压缩函数所需的全部的K[ ]数组。NIST声称它们是前32个素数的立方根的32位小数部分。例如，$2^{1/3}$的小数部分为0.2599210498948731647672106072782，它乘以2^{32}就是1116352408.8404644807431890114033，并且四舍五入就可以产生表中的第一个条目0x428A2F98。

我们猜测NIST从未想过动态地生成K[ ]的值会是一种很好的办法。在他们的考虑当中，他们选择了这样一个奇怪的表生成函数，那么他们就可以声明表中没有“陷门”。

```
void sha256_init(sha256_state *md)
{
    md->S[0] = 0x6A09E667UL;
    md->S[1] = 0xBB67AE85UL;
    md->S[2] = 0x3C6EF372UL;
    md->S[3] = 0xA54FF53AUL;
    md->S[4] = 0x510E527FUL;
    md->S[5] = 0x9B05688CUL;
    md->S[6] = 0x1F83D9ABUL;
    md->S[7] = 0x5BE0CD19UL;
    md->buflen = md->msglen = 0;
}
```

这把状态初始化为默认的SHA-256状态。

```
static void sha256_compress(sha256_state *md)
{
   ulong32 W[64], a, b, c, d, e, f, g, h, t, t0, t1;
   unsigned x;

   /* load W[0..15] */
   for (x = 0; x < 16; x++) {
       LOAD32H(W[x], md->buf + 4 * x);
   }

   /* compute W[16...63] */
   for (x = 16; x < 64; x++) {
       W[x] = Gamma1(W[x - 2]) + W[x - 7] +
              Gamma0(W[x - 15]) + W[x - 16];
   }
```

此时，我们已经完全把消息分组扩展为W[0...63]，而且可以开始压缩数据了。注意其加法是模2^{32}的，但我们并不需要显式地去约简它们，因此轮函数只使用循环宏。类似于SHA-1，如果内存很紧张的话，我们可以动态地以及合适地扩展分组。

```
    /* load a copy of the state */
    a = md->S[0]; b = md->S[1]; c = md->S[2];
    d = md->S[3]; e = md->S[4]; f = md->S[5];
    g = md->S[6]; h = md->S[7];

    /* perform 64 rounds */
    for (x = 0; x < 64; x++) {
      t0 = h + Sigma1(e) + Ch(e, f, g) + K[x] + W[x];
      t1 = Sigma0(a) + Maj(a, b, c);
      d += t0;
      h  = t0 + t1;

      /* swap */
      t = h; h = g; g = f; f = e; e = d;
      d = c; c = b; b = a; a = t;
    }
```

像SHA-1实现一样，我们把循环完全凑在一起了。我们可以把它拆分为8次或者完全拆分以获得速度上的提升。完全拆分它也可以允许我们把K[ ]常量嵌入到代码中，这避免了在轮函数中的查表操作。拆分的好处取决于平台。在大多数的x86平台上，得到的很少（如果有的话）而且代码大小的增加很显著。

通过定义一个轮宏，我们能够很好地执行拆分。

```
#define RND(a,b,c,d,e,f,g,h,i)                          \
     t0 = h + Sigma1(e) + Ch(e, f, g) + K[i] + W[i];    \
     t1 = Sigma0(a) + Maj(a, b, c);                     \
     d += t0;                                           \
     h  = t0 + t1;

for (i = 0; i < 64; ) {
    RND(a,b,c,d,e,f,g,h,i); ++i;
    RND(h,a,b,c,d,e,f,g,i); ++i;
    RND(g,h,a,b,c,d,e,f,i); ++i;
    RND(f,g,h,a,b,c,d,e,i); ++i;
    RND(e,f,g,h,a,b,c,d,i); ++i;
    RND(d,e,f,g,h,a,b,c,i); ++i;
    RND(c,d,e,f,g,h,a,b,i); ++i;
    RND(b,c,d,e,f,g,h,a,i); ++i;
}
```

这将会执行SHA-256压缩而不用执行前一个程序中的所有的交换操作。注意在宏之后我们对*i*进行递增，因为我们在源代码中碰到一个顺序点（squence point）之前它会被多次使用。

```
    /* update state */
    md->S[0] += a;
    md->S[1] += b;
    md->S[2] += c;
    md->S[3] += d;
    md->S[4] += e;
    md->S[5] += f;
    md->S[6] += g;
    md->S[7] += h;
}
```

像SHA-1一样，我们把副本加到SHA-256状态中以结束压缩函数。

```
void sha256_process(     sha256_state *md,
                   const unsigned char *buf,
                         unsigned long  len)
{
   unsigned long x, y;

   while (len) {
       x    = (64 - md->buflen) < len ? 64 - md->buflen : len;
       len -= x;

       /* copy x bytes from buf to the buffer */
       for (y = 0; y < x; y++) {
           md->buf[md->buflen++] = *buf++;
       }

       if (md->buflen == 64) {
          sha256_compress(md);
          md->buflen  = 0;
          md->msglen += 64;
       }
   }
}
```

这是一个直接对SHA-1函数的复制，只是把sh1_compress函数换成sha256_compress。

提示 大多数流行的散列算法如MD4、MD5、SHA-1、SHA-256等的“process（处理）”函数是如此的相似，以至于可以使用一个单独的宏对给定的散列算法扩展成其所需的处理函数。

LibTomCrypt算法库使用了这种技术，因此对宏的优化（例如零复制散列）也同样用在所有使用这个宏的散列算法上。

```
void sha256_done(   sha256_state *md,
                 unsigned char *dst)
{
   ulong32 l1, l2, i;

   /* compute final length as 8*md->msglen */
   md->msglen += md->buflen;
   l2 = md->msglen >> 29;
   l1 = (md->msglen << 3) & 0xFFFFFFFF;

   /* add the padding bit */
   md->buf[md->buflen++] = 0x80;

   /* if the current len > 56 we have to finish this block */
   if (md->buflen > 56) {
      while (md->buflen < 64) {
          md->buf[md->buflen++] = 0x00;
      }
      sha256_compress(md);
      md->buflen = 0;
   }

   /* now pad until we are at pos 56 */
   while (md->buflen < 56) {
       md->buf[md->buflen++] = 0x00;
```

```
   }

   /* store the length */
   STORE32H(l2, md->buf + 56);
   STORE32H(l1, md->buf + 60);

   /* compress */
   sha256_compress(md);

   /* extract the state */
   for (i = 0; i < 8; i++) {
      STORE32H(md->S[i], dst + i*4);
   }
}
```

这个函数也是从SHA-1实现中复制过来的，改变了函数名并且我们保存的是8个32位字而不是5个。

```
void sha256_memory(const unsigned char *in,
                         unsigned long  len,
                         unsigned char *dst)
{
   sha256_state md;
   sha256_init(&md);
   sha256_process(&md, in, len);
   sha256_done(&md, dst);
}
```

我们的辅助函数如下。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(void)
{
  static const struct {
      char *msg;
      unsigned char hash[32];
  } tests[] = {
    { "abc",
      { 0xba, 0x78, 0x16, 0xbf, 0x8f, 0x01, 0xcf, 0xea,
        0x41, 0x41, 0x40, 0xde, 0x5d, 0xae, 0x22, 0x23,
        0xb0, 0x03, 0x61, 0xa3, 0x96, 0x17, 0x7a, 0x9c,
        0xb4, 0x10, 0xff, 0x61, 0xf2, 0x00, 0x15, 0xad }
    },
    { "abcdbcdecdefdefgefghfghighijhi"
      "jkijkljklmklmnlmnomnopnopq",
      { 0x24, 0x8d, 0x6a, 0x61, 0xd2, 0x06, 0x38, 0xb8,
        0xe5, 0xc0, 0x26, 0x93, 0x0c, 0x3e, 0x60, 0x39,
        0xa3, 0x3c, 0xe4, 0x59, 0x64, 0xff, 0x21, 0x67,
        0xf6, 0xec, 0xed, 0xd4, 0x19, 0xdb, 0x06, 0xc1 }
    },
  };
  int i;
  unsigned char tmp[32];

  for (i = 0; i < 2; i++) {
     sha256_memory((unsigned char *)tests[i].msg,
```

```
                    strlen(tests[i].msg), tmp);
        if (memcmp(tests[i].hash, tmp, 32)) {
            printf("Failed test %d\n", i);
            return EXIT_FAILURE;
        }
    }
    printf("SHA-256 Passed\n");
    return EXIT_SUCCESS;
}
```

我们的测试程序是用来确保我们的SHA-256实现可以正确地运行。

5.2.4 SHA-512的设计

SHA-512是在SHA-256算法之后设计的。它们在消息分组大小、轮常量、轮数以及Sigma/Gamma函数上是不同的。它是由一个有8个64位字的状态以及遵循与其他散列算法相同的扩展和压缩的工作流程所组成。SHA-512使用128个字节的分组，而不是SHA-1和SHA-256所使用的64个字节的分组。新的宏如下所示。

```
#define Sigma0(x)        (S(x, 28) ^ S(x, 34) ^ S(x, 39))
#define Sigma1(x)        (S(x, 14) ^ S(x, 18) ^ S(x, 41))
```

这些是轮函数宏，注意其中的循环和移位操作是面向64位的，而不是SHA-256中的32位。

```
#define Gamma0(x)        (S(x, 1) ^ S(x, 8) ^ R(x, 7))
#define Gamma1(x)        (S(x, 19) ^ S(x, 61) ^ R(x, 6))
```

这些是扩展函数宏，同样，它们也是64位的操作。

1. SHA-512的状态

SHA-512的状态是8个64位字，用S[0...7]来表示，初始时设置为如下的值。

```
S[0] = 0x6a09e667f3bcc908;
S[1] = 0xbb67ae8584caa73b;
S[2] = 0x3c6ef372fe94f82b;
S[3] = 0xa54ff53a5f1d36f1;
S[4] = 0x510e527fade682d1;
S[5] = 0x9b05688c2b3e6c1f;
S[6] = 0x1f83d9abfb41bd6b;
S[7] = 0x5be0cd19137e2179;
```

2. SHA-512的扩展

SHA-512的扩展和SHA-256的扩展工作相同，不同的是它使用的是64位的宏并且我们必须产生80个字。前16个64位字是从128个字节的消息分组中以big-endian格式加载而来的。接下来的64个字是由下面的循环生成的。

```
for (i = 16; i < 80; i++) {
   W[i] = Gamma1(W[i - 2]) + W[i - 7] +
          Gamma0(W[i - 15]) + W[i - 16];
}
```

3. SHA-512的压缩

SHA-512的压缩类似于SHA-256的压缩。对于轮函数，它使用同样的类型，不同的是它有

80轮而不是64轮。它也使用80个64位的常量字，用K[0...79]来表示，它们也是用和SHA-256算法中所用的64个字非常相似的方法衍生出来的。

4. SHA-512的实现

下面是SHA-512的实现，它衍生自SHA-256实现的源代码。它和SHA-1以及SHA-256有相同的调用约定，不同的是它产生一个64个字节的消息摘要而不是20或者30个字节的消息摘要。

sha512.c:

```
#ifdef _MSC_VER
   #define CONST64(n) n ## ui64
   typedef unsigned __int64 ulong64;
#else
   #define CONST64(n) n ## ULL
   typedef unsigned long long ulong64;
#endif
```

首先，我们需要一种使用64位数据类型的方法。对于SHA-512算法的大部分来说，它是适合于用64位字的SHA-256的而不是32位字。在C99中，一个至少为64位的数据类型定义为*unsigned long long*，它可以很好地工作于大部分的UNIX CC和GNU CC。不幸的是，微软的处理器不支持C99而且实现起来也不相同。

我们的CONST64宏给了我们一种合理地可移植的定义64位常量的方法。我们的ulong64类型定义可以让我们以一种有效的方式来使用64位字。

```
/* Helpful macros */
#define STORE64H(x, y)                                    \
   { (y)[0] = (unsigned char)(((x)>>56)&255);              \
     (y)[1] = (unsigned char)(((x)>>48)&255);              \
     (y)[2] = (unsigned char)(((x)>>40)&255);              \
     (y)[3] = (unsigned char)(((x)>>32)&255);              \
     (y)[4] = (unsigned char)(((x)>>24)&255);              \
     (y)[5] = (unsigned char)(((x)>>16)&255);              \
     (y)[6] = (unsigned char)(((x)>>8)&255);               \
     (y)[7] = (unsigned char)((x)&255); }

#define LOAD64H(x, y)                            \
   { x = (((ulong64)((y)[0] & 255))<<56)|  \
         (((ulong64)((y)[1] & 255))<<48)|  \
         (((ulong64)((y)[2] & 255))<<40)|  \
         (((ulong64)((y)[3] & 255))<<32)|  \
         (((ulong64)((y)[4] & 255))<<24)|  \
         (((ulong64)((y)[5] & 255))<<16)|  \
         (((ulong64)((y)[6] & 255))<<8)|   \
         (((ulong64)((y)[7] & 255))); }

#define ROR(x, y) \
((((ulong64)(x)>>(ulong64)((y)&63)) |  \
(((ulong64)(x)&CONST64(0xFFFFFFFFFFFFFFFF))<< \
(ulong64)(64-((y)&63)))) & CONST64(0xFFFFFFFFFFFFFFFF))
```

这些是我们的友好的宏，它们适用于64位的数据类型。最好把这些定义放到一个共享的头文件中。

```
#define Ch(x,y,z)    (z ^ (x & (y ^ z)))
#define Maj(x,y,z)   (((x | y) & z) | (x & y))
```

```
#define S(x, n)         ROR((x),(n))
#define R(x, n)         (((x)&CONST64(0xFFFFFFFFFFFFFFFF)) \
                         >>((ulong64)n))
#define Sigma0(x)       (S(x, 28) ^ S(x, 34) ^ S(x, 39))
#define Sigma1(x)       (S(x, 14) ^ S(x, 18) ^ S(x, 41))
#define Gamma0(x)       (S(x, 1) ^ S(x, 8) ^ R(x, 7))
#define Gamma1(x)       (S(x, 19) ^ S(x, 61) ^ R(x, 6))
```

这些是对于512位散列的SHA宏。注意这里我们执行的是64位的操作（包括移位和循环）。

```
typedef struct {
   unsigned char buf[128];
   unsigned long buflen, msglen;
   ulong64       S[8];
} sha512_state;
```

这是我们的SHA-512的状态。注意SHA-512使用一个128个字节的分组，因为我们的缓冲区现在更大了。仍然有8个链接变量（chaining variables），但它们现在是64位的。

```
static const ulong64 K[80] = {
CONST64(0x428a2f98d728ae22), CONST64(0x7137449123ef65cd),
CONST64(0xb5c0fbcfec4d3b2f), CONST64(0xe9b5dba58189dbbc),
CONST64(0x3956c25bf348b538), CONST64(0x59f111f1b605d019),
CONST64(0x923f82a4af194f9b), CONST64(0xab1c5ed5da6d8118),
CONST64(0xd807aa98a3030242), CONST64(0x12835b0145706fbe),
CONST64(0x243185be4ee4b28c), CONST64(0x550c7dc3d5ffb4e2),
CONST64(0x72be5d74f27b896f), CONST64(0x80deb1fe3b1696b1),
CONST64(0x9bdc06a725c71235), CONST64(0xc19bf174cf692694),
CONST64(0xe49b69c19ef14ad2), CONST64(0xefbe4786384f25e3),
CONST64(0x0fc19dc68b8cd5b5), CONST64(0x240ca1cc77ac9c65),
CONST64(0x2de92c6f592b0275), CONST64(0x4a7484aa6ea6e483),
CONST64(0x5cb0a9dcbd41fbd4), CONST64(0x76f988da831153b5),
CONST64(0x983e5152ee66dfab), CONST64(0xa831c66d2db43210),
CONST64(0xb00327c898fb213f), CONST64(0xbf597fc7beef0ee4),
CONST64(0xc6e00bf33da88fc2), CONST64(0xd5a79147930aa725),
CONST64(0x06ca6351e003826f), CONST64(0x142929670a0e6e70),
CONST64(0x27b70a8546d22ffc), CONST64(0x2e1b21385c26c926),
CONST64(0x4d2c6dfc5ac42aed), CONST64(0x53380d139d95b3df),
CONST64(0x650a73548baf63de), CONST64(0x766a0abb3c77b2a8),
CONST64(0x81c2c92e47edaee6), CONST64(0x92722c851482353b),
CONST64(0xa2bfe8a14cf10364), CONST64(0xa81a664bbc423001),
CONST64(0xc24b8b70d0f89791), CONST64(0xc76c51a30654be30),
CONST64(0xd192e819d6ef5218), CONST64(0xd69906245565a910),
CONST64(0xf40e35855771202a), CONST64(0x106aa07032bbd1b8),
CONST64(0x19a4c116b8d2d0c8), CONST64(0x1e376c085141ab53),
CONST64(0x2748774cdf8eeb99), CONST64(0x34b0bcb5e19b48a8),
CONST64(0x391c0cb3c5c95a63), CONST64(0x4ed8aa4ae3418acb),
CONST64(0x5b9cca4f7763e373), CONST64(0x682e6ff3d6b2b8a3),
CONST64(0x748f82ee5defb2fc), CONST64(0x78a5636f43172f60),
CONST64(0x84c87814a1f0ab72), CONST64(0x8cc702081a6439ec),
CONST64(0x90befffa23631e28), CONST64(0xa4506cebde82bde9),
CONST64(0xbef9a3f7b2c67915), CONST64(0xc67178f2e372532b),
CONST64(0xca273eceea26619c), CONST64(0xd186b8c721c0c207),
CONST64(0xeada7dd6cde0eb1e), CONST64(0xf57d4f7fee6ed178),
CONST64(0x06f067aa72176fba), CONST64(0x0a637dc5a2c898a6),
CONST64(0x113f9804bef90dae), CONST64(0x1b710b35131c471b),
CONST64(0x28db77f523047d84), CONST64(0x32caab7b40c72493),
CONST64(0x3c9ebe0a15c9bebc), CONST64(0x431d67c49c100d4c),
```

```
CONST64(0x4cc5d4becb3e42b6), CONST64(0x597f299cfc657e2a),
CONST64(0x5fcb6fab3ad6faec), CONST64(0x6c44198c4a475817)
};
```

这是用于压缩函数的80个64位的轮常量。注意我们为了可移植而使用了CONST64宏。

```
void sha512_init(sha512_state *md)
{
    md->S[0] = CONST64(0x6a09e667f3bcc908);
    md->S[1] = CONST64(0xbb67ae8584caa73b);
    md->S[2] = CONST64(0x3c6ef372fe94f82b);
    md->S[3] = CONST64(0xa54ff53a5f1d36f1);
    md->S[4] = CONST64(0x510e527fade682d1);
    md->S[5] = CONST64(0x9b05688c2b3e6c1f);
    md->S[6] = CONST64(0x1f83d9abfb41bd6b);
    md->S[7] = CONST64(0x5be0cd19137e2179);
    md->buflen = md->msglen = 0;
}
```

这个函数初始化SHA-512状态为默认的状态。

```
static void sha512_compress(sha512_state *md)
{
   ulong64 W[80], a, b, c, d, e, f, g, h, t, t0, t1;
   unsigned x;

   /* load W[0..15] */
   for (x = 0; x < 16; x++) {
       LOAD64H(W[x], md->buf + 8 * x);
   }

   /* compute W[16...80] */
   for (x = 16; x < 80; x++) {
       W[x] = Gamma1(W[x - 2]) + W[x - 7] +
              Gamma0(W[x - 15]) + W[x - 16];
   }
```

此时，我们已把128个字节的消息输入扩展为80个64位字。像SHA-1和SHA-256函数一样，我们可以动态地计算扩展字。

```
   /* load a copy of the state */
   a = md->S[0]; b = md->S[1]; c = md->S[2];
   d = md->S[3]; e = md->S[4]; f = md->S[5];
   g = md->S[6]; h = md->S[7];

   /* perform 80 rounds */
   for (x = 0; x < 80; x++) {
     t0 = h + Sigma1(e) + Ch(e, f, g) + K[x] + W[x];
     t1 = Sigma0(a) + Maj(a, b, c);
     d += t0;
     h  = t0 + t1;

     /* swap */
     t = h; h = g; g = f; f = e; e = d;
     d = c; c = b; b = a; a = t;
   }
```

它执行了压缩轮函数。我们可以把它拆分成8次或者完全拆分。通常情况下，8次会得到一

个可变的性能提升，而完全拆分的回报不多。在AMD64系列上，完全拆分并没提高性能反而浪费了缓存空间。

```
   /* update state */
   md->S[0] += a;
   md->S[1] += b;
   md->S[2] += c;
   md->S[3] += d;
   md->S[4] += e;
   md->S[5] += f;
   md->S[6] += g;
   md->S[7] += h;
}

void sha512_process(      sha512_state *md,
                    const unsigned char *buf,
                          unsigned long  len)
{
   unsigned long x, y;

   while (len) {
       x     = (128 - md->buflen) < len ? 128 - md->buflen : len;
       len -= x;

       /* copy x bytes from buf to the buffer */
       for (y = 0; y < x; y++) {
           md->buf[md->buflen++] = *buf++;
       }

       if (md->buflen == 128) {
          sha512_compress(md);
          md->buflen  = 0;
          md->msglen += 128;
       }
   }
}
```

这个处理函数类似于SHA-1和SHA-256中的处理函数,不同的是它使用了128个字节的分组。正如在其他算法中一样，我们也可以在这里使用一个零复制机制，并且强烈推荐这么做。

```
void sha512_done(   sha512_state *md,
                 unsigned char *dst)
{
   ulong64 l1, l2, i;

   /* compute final length as 8*md->msglen */
   md->msglen += md->buflen;
   l2 = md->msglen >> 29;
   l1 = (md->msglen << 3) & 0xFFFFFFFF;

   /* add the padding bit */
   md->buf[md->buflen++] = 0x80;

   /* if the current len > 112 we have to finish this block */
   if (md->buflen > 112) {
      while (md->buflen < 128) {
          md->buf[md->buflen++] = 0x00;
```

```
        }
        sha512_compress(md);
        md->buflen = 0;
     }

     /* now pad until we are at pos 112 */
     while (md->buflen < 112) {
         md->buf[md->buflen++] = 0x00;
     }

     /* store the length */
     STORE64H(l2, md->buf + 112);
     STORE64H(l1, md->buf + 120);

     /* compress */
     sha512_compress(md);

     /* extract the state */
     for (i = 0; i < 8; i++) {
       STORE64H(md->S[i], dst + i*8);
     }
}
```

这个函数结束散列并输出摘要。注意我们必须存储一个128位的长度。因为我们的分组大小是128个字节，因此我们必须填充直到消息的长度是112模128个字节。

```
void sha512_memory(const unsigned char *in,
                         unsigned long  len,
                         unsigned char *dst)
{
   sha512_state md;
   sha512_init(&md);
   sha512_process(&md, in, len);
   sha512_done(&md, dst);
}
```

这是我们所熟悉的辅助函数，它用于执行对一个内存缓冲区的SHA-512压缩。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(void)
{
  static const struct {
      char *msg;
      unsigned char hash[64];
  } tests[] = {
    { "abc",
     { 0xdd, 0xaf, 0x35, 0xa1, 0x93, 0x61, 0x7a, 0xba,
       0xcc, 0x41, 0x73, 0x49, 0xae, 0x20, 0x41, 0x31,
       0x12, 0xe6, 0xfa, 0x4e, 0x89, 0xa9, 0x7e, 0xa2,
       0x0a, 0x9e, 0xee, 0xe6, 0x4b, 0x55, 0xd3, 0x9a,
       0x21, 0x92, 0x99, 0x2a, 0x27, 0x4f, 0xc1, 0xa8,
       0x36, 0xba, 0x3c, 0x23, 0xa3, 0xfe, 0xeb, 0xbd,
       0x45, 0x4d, 0x44, 0x23, 0x64, 0x3c, 0xe8, 0x0e,
       0x2a, 0x9a, 0xc9, 0x4f, 0xa5, 0x4c, 0xa4, 0x9f }
    },
    { "abcdefghbcdefghicdefghijdefghijkefghijklfghijkl"
```

```
        "mghijklmnhijklmnoijklmnopjklmnopqklmnopqrlmnopq"
        "rsmnopqrstnopqrstu",
       { 0x8e, 0x95, 0x9b, 0x75, 0xda, 0xe3, 0x13, 0xda,
         0x8c, 0xf4, 0xf7, 0x28, 0x14, 0xfc, 0x14, 0x3f,
         0x8f, 0x77, 0x79, 0xc6, 0xeb, 0x9f, 0x7f, 0xa1,
         0x72, 0x99, 0xae, 0xad, 0xb6, 0x88, 0x90, 0x18,
         0x50, 0x1d, 0x28, 0x9e, 0x49, 0x00, 0xf7, 0xe4,
         0x33, 0x1b, 0x99, 0xde, 0xc4, 0xb5, 0x43, 0x3a,
         0xc7, 0xd3, 0x29, 0xee, 0xb6, 0xdd, 0x26, 0x54,
         0x5e, 0x96, 0xe5, 0x5b, 0x87, 0x4b, 0xe9, 0x09 }
      },
   };
   int i;
   unsigned char tmp[64];
   for (i = 0; i < 2; i++) {
      sha512_memory((unsigned char *)tests[i].msg,
                    strlen(tests[i].msg), tmp);
      if (memcmp(tests[i].hash, tmp, 64)) {
          printf("Failed test %d\n", i);
          return EXIT_FAILURE;
      }
   }
   printf("SHA-512 Passed\n");
   return EXIT_SUCCESS;
}
```

这是我们的演示程序，它也可以测试这个实现。我们只实现了3个NIST标准测试向量中的前两个。

5.2.5 SHA-224的设计

SHA-224是一个产生224位消息摘要的散列算法，它使用SHA-256来执行散列。SHA-224以使用8个不同的字列来初始化其状态。

```
S[0] = 0xc1059ed8;
S[1] = 0x367cd507;
S[2] = 0x3070dd17;
S[3] = 0xf70e5939;
S[4] = 0xffc00b31;
S[5] = 0x68581511;
S[6] = 0x64f98fa7;
S[7] = 0xbefa4fa4;
```

SHA-224对所有的消息分组都使用SHA-256的扩展和压缩函数。当SHA-256的最终摘要产生时，只复制前28个字节给调用函数。

从性能的角度来看，SHA-224不比SHA-256快，而且从安全性的角度来看，它也没有SHA-256安全。在某些时候截短一个散列会是个不错的方法，正如我们稍后将在数字签名中看到的，但是还有比修改一个现有的散列算法更为简单的方法来完成这件事。

如果你的应用程序使用了ECC-224，那么SHA-224会是一个很有用处的选择，因为它所产生的输出和曲线的阶——将在第9章中介绍的一个有用的性质——几乎是相同的。

为完整起见，下面是对SHA-224的测试向量，类似于SHA-256。

```
static const struct {
```

```
        char *msg;
        unsigned char hash[28];
    } tests[] = {
      { "abc",
        { 0x23, 0x09, 0x7d, 0x22, 0x34, 0x05, 0xd8,
          0x22, 0x86, 0x42, 0xa4, 0x77, 0xbd, 0xa2,
          0x55, 0xb3, 0x2a, 0xad, 0xbc, 0xe4, 0xbd,
          0xa0, 0xb3, 0xf7, 0xe3, 0x6c, 0x9d, 0xa7 }
      },
      { "abcdbcdecdefdefgefghfghighijh"
        "ijkijkljklmklmnlmnomnopnopq",
        { 0x75, 0x38, 0x8b, 0x16, 0x51, 0x27, 0x76,
          0xcc, 0x5d, 0xba, 0x5d, 0xa1, 0xfd, 0x89,
          0x01, 0x50, 0xb0, 0xc6, 0x45, 0x5c, 0xb4,
          0xf5, 0x8b, 0x19, 0x52, 0x52, 0x25, 0x25 }
      },
    };
```

5.2.6 SHA-384的设计

类似于SHA-224，SHA-384也是在一个SHS算法之后设计的。在本例中，SHA-384是基于SHA-512的。像SHA-224一样，我们也以一个修改了的状态开始。

```
S[0] = CONST64(0xcbbb9d5dc1059ed8);
S[1] = CONST64(0x629a292a367cd507);
S[2] = CONST64(0x9159015a3070dd17);
S[3] = CONST64(0x152fecd8f70e5939);
S[4] = CONST64(0x67332667ffc00b31);
S[5] = CONST64(0x8eb44a8768581511);
S[6] = CONST64(0xdb0c2e0d64f98fa7);
S[7] = CONST64(0x47b5481dbefa4fa4);
```

SHA-384使用SHA-512算法来执行实际的消息散列。SHA-384的输出是截短的SHA-512的输出，是把其前48个字节复制给调用函数。当和ECC-384曲线一起工作时，SHA-384会是一个有用的散列算法的选择。

为完整起见，下面是SHA-384的测试向量。

```
static const struct {
    char *msg;
    unsigned char hash[48];
} tests[] = {
  { "abc",
    { 0xcb, 0x00, 0x75, 0x3f, 0x45, 0xa3, 0x5e, 0x8b,
      0xb5, 0xa0, 0x3d, 0x69, 0x9a, 0xc6, 0x50, 0x07,
      0x27, 0x2c, 0x32, 0xab, 0x0e, 0xde, 0xd1, 0x63,
      0x1a, 0x8b, 0x60, 0x5a, 0x43, 0xff, 0x5b, 0xed,
      0x80, 0x86, 0x07, 0x2b, 0xa1, 0xe7, 0xcc, 0x23,
      0x58, 0xba, 0xec, 0xa1, 0x34, 0xc8, 0x25, 0xa7 }
  },
  { "abcdefghbcdefghicdefghijdefghi"
    "jkefghijklfghijklmghijklmnhijk"
    "lmnoijklmnopjklmnopqklmnopqrlm"
    "nopqrsmnopqrstnopqrstu",
    { 0x09, 0x33, 0x0c, 0x33, 0xf7, 0x11, 0x47, 0xe8,
```

```
        0x3d, 0x19, 0x2f, 0xc7, 0x82, 0xcd, 0x1b, 0x47,
        0x53, 0x11, 0x1b, 0x17, 0x3b, 0x3b, 0x05, 0xd2,
        0x2f, 0xa0, 0x80, 0x86, 0xe3, 0xb0, 0xf7, 0x12,
        0xfc, 0xc7, 0xc7, 0x1a, 0x55, 0x7e, 0x2d, 0xb9,
        0x66, 0xc3, 0xe9, 0xfa, 0x91, 0x74, 0x60, 0x39 }
  },
};
```

5.2.7 零复制散列

先前我们间接地提到了用一种叫做零复制的技术来提高散列算法的吞吐量。我们现在将对这种用于SHA-1 *process*函数中的技术进行介绍。我们只介绍其修改过的代码以节省空间。

sha1zc.c:

```
static void sha1_compress(         sha1_state *md,
                           const unsigned char *buf)
{
   ulong32 W[80], a, b, c, d, e, t;
   unsigned x;

   /* load W[0..15] */
   for (x = 0; x < 16; x++) {
       LOAD32H(W[x], buf + 4 * x);
   }
```

我们首先要修改压缩函数以接受来自调用函数的消息分组，而不是隐式地来自SHA-1状态。

```
void sha1_process(         sha1_state *md,
                   const unsigned char *buf,
                         unsigned long  len)
{
   unsigned long x, y;

   while (len) {
       /* zero copy 64 byte chunks */
       while (len >= 64 && md->buflen == 0) {
          sha1_compress(md, buf);
          buf        += 64;
          md->msglen += 64;
          len        -= 64;
       }

       x    = (64 - md->buflen) < len ? 64 - md->buflen : len;
       len -= x;

       /* copy x bytes from buf to the buffer */
       for (y = 0; y < x; y++) {
           md->buf[md->buflen++] = *buf++;
       }

       if (md->buflen == 64) {
          sha1_compress(md, md->buf);
          md->buflen  = 0;
          md->msglen += 64;
       }
   }
}
```

这是我们新的修改过了的处理函数。在内循环里，我们执行了零复制优化，通过把64字节的分组直接传给压缩函数，而没有首先把它复制到内部状态中。只有当SHA-1的状态是空（buflen为0）并且至少还剩下64个字节的消息需要处理时才能这么做。

这种优化似乎在一个带有大的L1数据缓存的处理器上不是很明显，例如AMD Opteron。但是，在更多的紧凑型的处理器上，例如ARM系列，额外的数据移动被取消了，即一个16MHz的数据总线意味着一种真正的性能损失。

```
void sha1_done(   sha1_state *md,
                  unsigned char *dst)
{
   ulong32 l1, l2, i;

   /* compute final length as 8*md->msglen */
   md->msglen += md->buflen;
   l2 = md->msglen >> 29;
   l1 = (md->msglen << 3) & 0xFFFFFFFF;

   /* add the padding bit */
   md->buf[md->buflen++] = 0x80;

   /* if the current len > 56 we have to finish this block */
   if (md->buflen > 56) {
      while (md->buflen < 64) {
         md->buf[md->buflen++] = 0x00;
      }
      sha1_compress(md, md->buf);
      md->buflen = 0;
   }

   /* now pad until we are at pos 56 */
   while (md->buflen < 56) {
       md->buf[md->buflen++] = 0x00;
   }

   /* store the length */
   STORE32H(l2, md->buf + 56);
   STORE32H(l1, md->buf + 60);

   /* compress */
   sha1_compress(md, md->buf);

   /* extract the state */
   for (i = 0; i < 5; i++) {
      STORE32H(md->S[i], dst + i*4);
   }
}
```

为完整起见，这是修改后使用了新的压缩函数调用的done（完成）函数。

5.3 PKCS #5 密钥衍生

散列函数可以用于解决许多不同的问题，从完整性校验和认证（参见第6章）到伪随机数生成（参见第3章）和密钥衍生。现在我们将要研究后面一种性质。

密钥衍生函数（Key Derivation Function，KDF）从其他的熵源衍生出密钥，同时会保留输

入的熵而且是单向的。密钥衍生不仅仅是常用于生成密钥，它也用于衍生初始值（IV）以及*nonce*（参见第6章）。密钥衍生函数的典型用法是，取一个秘密例如口令或者共享秘密（参见第9章）和盐渍来生成密钥和IV。当会话第一次创建时，盐渍是随机生成的以防止字典攻击。它和使用随机的秘密不是同样重要的，因为在这种情况下字典攻击不会有效。

PKCS #5（ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-5v2/pkcs5v2-0.pdf）是由其创建者RSA Data Securtiy Corporation提出的公钥密码标准系列中的一种。它定义了两个KDF算法，分别叫做PBKDF1和PBKDF2。前者是一个旧的标准，原准备用于DES的，因此它有一个固定大小的输入。PBKDF2更加灵活而且也是PKCS #5标准所推荐的KDF。

我们将要介绍的算法使用了一个称为HMAC（Hash Message Authentication Code，散列消息认证码）的算法，我们还没有介绍过它。鼓励不懂的读者在看这个算法（如图5-6所示）之前先看一下第6章中对HMAC的介绍。

输入：

secret: 作为主密钥（master key）来使用的秘密，衍生为会话密钥（session key）
salt: 随机的非秘密的字符串，用以防止字典攻击
iterations：主循环中的迭代次数
w: 所用散列的摘要大小
outlen: 需要的KDF数据的数量

输出：

out: KDF数据

1. for blkNo from 1 to ceil(*outlen*/*w*) do
    1. T = HMAC(*secret*, *salt* || blkNo)
    2. U = T
    3. for i from 1 to *iterations* do
        i. T = HMAC(*secret*, T)
        ii. U = U xor T
    4. *out* = *out* || U
2. Truncate *out* to *outlen* bytes

图5-6 PKCS #5 PBKDF2算法

*blkNo*的值作为一个32位的big-endian数附加到盐渍后面。这个算法以计算附带*blkNo*值的盐渍的HMAC而开始。这给出了用于传给算法的起始值。我们把这个值复制到U，然后在每一次迭代中重复地对T的值进行HMAC，并把输出和U进行异或（XOR）。迭代计数器的作用是使得字典和穷举搜索攻击更慢。例如，如果它的值为1 024，那么实际上就相当于给秘密密钥又增加了10位。

这个函数所生成的数据即可以用于密钥也可以用于会话初始值，例如IV和nonce。例如，为了用CTR模式使用AES-128，调用函数可以使用这个KDF来生成32个字节的数据。前16个字节可以作为AES的密钥，后16个字节可以作为CTR的初始值。如果我们使用SHA-256作为HMAC

的散列算法，那么我们就只需要循环一次，因为在第4步已经生成了所需的32个字节。例如，假如我们使用SHA-1，那么就要循环两次，产生的40个字节会被截短而只剩下32个字节。

这个函数要求秘密不能被攻击者所预测。盐渍应用是随机的，但不能是一个秘密，因此接收者也需要知道它已生成同样的会话值。盐渍可以是任何长度，在实际应用中，它应该不比*secret*大而且不能小于8个字节。

对于想要了解更多内容的读者也许想把常被弄错的散列构造（secret ‖ data）作为一个像消息认证一样的原型来使用。但是，我们将在第6章中看到，这不是一个安全的结构并且应该避免。当它应用于PKCS #5，它也许很安全；但是，出于交互性目的来考虑，人们应该选择使用一个合适的HMAC构造。

5.4 总结

正如我们已看到的，SHS算法是很容易实现的，也正如我们提到的，也可以很方便地获得它们的实现。我们现在要考虑的是更加具体地在实际系统中使用散列算法的例子。即它们不能用来做哪些事，怎样对它们进行空间和时间上的优化，最后给出一个具体的使用前面的AES CTR示例（参见第4章）来应用PKCS #5的例子。

如果我们能记住散列算法是一种伪随机函数（PRF），那么利用它们来做它们应该能做的事是很容易的。就像分组密码是伪随机置换（PRP）而且它们也有自己的使用限制，散列算法也一样。

5.4.1 散列算法可以做哪些事

散列算法是伪随机函数。意思是指它们伪随机地把输入映射到输出。不像分组密码，它们是PRP，而散列算法的输入（定义域）和输出（对应域）的大小是不相同的。典型的散列算法可以允许大到$2^{64}-1$个位并且产生数百位的固定大小的输出。伪随机映射要有一种有用的特性，即单向性。如果映射是很难控制的话，它也要求具有碰撞约束的能力。

1. 单向性

如果一个函数，从一个方向来计算是简单的，而从其逆方向来计算是很困难的，那么就称这个函数是单向的。从散列函数方面来看，意思是指消息摘要是很容易计算出来的。而找到产生这个消息摘要的消息是很困难的。

2. 口令

与用户口令和密码短语（pass phrase）一起工作是使用这个性质的一个例子。理想情况下，一个要求用户对自身进行认证的网络应用程序会要求一个证书，例如口令。用明文来发送口令意味着攻击者也能读取它。一个多用户环境下的口令列表（例如校园网）可能需要保存某些东西以和用户的证书进行比较。散列算法就提供了一种可以证明证书的所有权又不会泄露证书自身的方法。

3. 随机数生成器

另外一种采用了这种性质的有用的任务是随机数生成器，以及伪随机变体。通过对种子数

据进行散列，其输出就不会泄露生成器秘密参与的种子数据。这个种子数据常常含有隐私信息，例如网络流量或者按键。

4. 碰撞约束

为了能够应用在密码学中，散列算法必须是碰撞约束的。通常，这个性质用于数字签名（参见第9章）的处理过程中。典型的公钥签名算法由于大量的循环而比典型的散列函数要慢。为了优化签名处理过程，加密系统可以选择对消息的散列进行签名而不是时整个消息自身。

这个过程实际上可以和对消息自身进行签名同样安全，但要求散列算法要有两种性质。它必须是碰撞约束的，而且它必须能产生各种大小的摘要。回忆一下在本章前面所讨论的生日悖论。如果你正在使用一个公钥算法，它需要2^{80}的工作（“工作”常指需要运算的数量；这里的工作可以说是公钥运算或者散列运算）来破解，你的散列应该产生不小于160位的消息摘要。

5. 文件清单（File Manifests）

碰撞约束在完整性校验的任务中也很有用。在大部分好的情况下，计算文件的消息摘要可以提供下载帮助以确保用户正确地下载文件。当然，这是假设在没有攻击者的情况之下的。有一个经常被提到的防卫，即由于消息摘要会保存在一台安全的服务器上，所以攻击者将不得不在那里修改它。但这是不正确的。一个攻击者只要在客户和服务器之间就可以实施攻击。这远比攻破一台安全的服务器简单得多。

6. 入侵检测（Intrusion Detection）

散列算法也用于入侵检测系统（Intrusion Detection Software，IDS）中，它扫描文件的修改，这是入侵者的标志。它们计算可执行文件的消息摘要，并且动态地与预存的值进行比较。这可以有效地工作，但要求IDS软件本身没有被利用。但是，人们不可能时刻都是警惕着的。在2005年，MD5（用于这项任务中的最常用的散列）被攻破。许多人，包括Dan Kaminsky，都支持特洛伊负载攻击（trojan payload attack）。

他们的攻击即灵活又有效。他们生成一个附带两个负载的程序，然后，插入一组将会和MD5相碰撞的数据，就能够把原来的分组交换出去，不过这要看他们想不想把负载激活了。这个程序首先检查包含的分组并根据不同的情况做不同的事情。两个文件可能有相同的MD5消息摘要，但它们可能行为非常不同。

许多Linux发行版本都转为使用多种散列算法来计算结果。例如，Gentoo Linux会检查它的“货物仓库”中的所有文件的MD5、RMD-160和SHA-256。虽然这在最大程度上可以和完美的256位的散列同样强（我们不会假设SHA-256是完美的），但一旦SHA-256被攻破的话，它仍然是很坚固的，至少检查RMD-160散列是可靠的［（RMD（即RIPEMD）代表RIPE 消息摘要，其中RIPE（RACE Integrity Primitives Evaluation）是一个欧洲标准。最初的RIPEMD算法一般认为是不够强的而且在2004年被攻破。RIPEMD-160、256和320算法大都基于MD4设计而且也不是特别有效］。

5.4.2 散列算法不能用来做哪些事

我们已经看到了一些明显的散列算法可以用来做事情的例子。也有许多突然出现的任务，

而且在实际系统中被错误地使用。产生这些问题的部分原因是基于这样一个事实，即人们认为散列算法是非常不可思议的函数。他们常常忘记我们能够在前面加上数据、追加数据以及预计算之类的东西。

1. 未盐渍化的口令（Unsalted Passwords）

这是加点盐就能使你更加健康（抱歉，我们无法忍受这种双关语）的情况之一。我们将在稍后更为详细地看到（快要到那儿了），口令散列是一种技巧活。假设我们想保存你的口令以备将来的比较使用。

显然，我们不能存储口令自身，因为一个能看到口令列表的攻击者会侵入系统。我们脑子里闪现出来的逻辑上的解决方法是对口令进行散列。毕竟它是单向的。什么情况下可能会出错?

2. 散列会形成不好的分组密码

把一个散列转变成一个分组密码通常是让人很感兴趣的。如果你仔细地观察散列算法的话，会发现它们实际上是伪装的分组密码。例如，SHACAL是一个使用SHA-1的160位的分组密码。它把W[ ]数组看成是密码的密钥，把状态看成是明文（或者密文）。你也可以用CTR模式的散列来创建一个流密码。

虽然看起来技术上是可行的而且似乎这样做也是安全的，但它们都不是标准的一部分。它们也比分组密码要慢。例如，在一个AMD Opteron处理器上MD5每个字节的散列将需要8个时钟周期。意思是说，输入的每个字节是8个周期而不是输出。MD5是64个字节的分组，因此，其压缩至少要花费512个时钟周期，这能够以每个字节32个时钟周期的速率来产生一个密钥流。AES CTR每个字节只需要稍微超过17个的时钟周期。

对于其他的散列算法，情况也并不乐观。在Opteron上，SHA-512输入的每个字节需要12个时钟周期，它将转变为输出的每个字节所需要的24个时钟周期。在Intel Pentium 4处理器上（为了举一个在其他处理器上的例子），SHA-1的每个字节需要18个时钟周期，转化为输出就是每个字节需要58个时钟周期（比在这个处理器上AES要慢两倍多）。

标准的缺乏以及这样做效率很差的事实通常使得用散列算法来实际分组密码是一种很糟糕的想法。它惟一有意义的地方就是，在你被软件或者硬件的空间所限的情况下。

3. 散列函数不是MAC

散列函数不是消息认证码算法。最常见的并且实际上是不安全的MAC的构造如下。

tag:=hash(*key*||*message*)

对这种构造的攻击是利用了这样一种事实，即攻击不需要以标准指定的同样的初始值开始。在接下来的处理过程中，我们实际上是对消息分组进行散列，即密钥、消息、MD强化。这产生了一个标记（tag），实质上它是在散列完所有数据之后的散列状态。攻击者可以追加一个分组（然后是另外一个MD5强化分组）并且把这个标记作为初始值。计算出来的散列将看起来很像是来自一个带有密钥的受害者的有效消息。

经常使用如下的组合来进行调整。

tag:=hash(hash(*key*||*message*))

这不会泄露内散列的内部状态。但是，它违背了MAC算法的一个目标。在离线的环境下它

是可被攻击的，我们将会在第6章的HMAC的讨论中看到。

4. 散列不能加倍

一种常见的让消息摘要的大小加倍的技巧是，把两个稍微有些不同的消息连接起来。例如：

digest:=hash(1||*message*)||hash(2||*message*)

这种技术的一个问题是，第一个消息分组的碰撞将导致后面所有分组的碰撞。例如，考虑其输入为1 || *block* || *message*，其中1 || *block*是散列中的一个消息分组（例如，对于SHA-1是64个字节）。如果我们能够找到一个*block*使得hash（1 || *block*）等于hash（2 || *block*），那么我们只要找到两个消息值使得散列的其余部分都发生碰撞就可以了。

5. 散列不能组合

加倍的技巧是失败了，但接着又有了一种想法，即把两个来自不同散列的消息摘要连接在一起。例如：

digest:= hash1(*message*) || hash2(*message*)

不幸地是，这没有组合中强度最大的散列好——至少不是使用典型的散列结构。假设在hash1中M和M'是碰撞的。那么，M || Q和M'|| Q也会碰撞，因为它们的散列状态在M（M'，分别地）之后碰撞。这样，攻击者只要能找到一对M和M'，然后继续攻击下一个散列。可以想象，两个合理的构造了的散列组合成一对能提供更高的安全性，但是，如果用的是同样的MD体系的散列，那么就不是这样了。

5.4.3 和口令一起工作

我们已提到过，当散列算法和口令一起工作的时候，它们表现还不错。它们不允许由输出确定输入，而且是碰撞约束的。实际上，如果用户能够产生正确的消息摘要，那么惟一的机会就是他们知道正确的输入。

安全地使用这种不错的性质是一种技巧上的挑战。首先，让我们来考虑离线的世界，然后我们会探索在线的世界。

1. 离线的口令

离线的口令校验是你在账号登录（在你的本地机器上）时所发现的那种情形。其目的是把你所产生的和存储在本地的证据进行比较。该证据只能由某些拥有所需要的证书的人来生成。

在离线的世界中，我们都是在和带有优先级的进程打交道，例如登录（尤其是在类UNIX平台上），因此这个证据生成进程是不会被篡改的。

如果我们简单地保存口令的散列，就可以防止攻击者直接找出口令。但不幸地是，我们不能阻止一种实用且很有效的攻击——字典攻击。用户想要取一些容易记忆的口令和密码短语。这意味着他们会取字典上面所包含的单词，而且有时候会把两个或多个混合在一起。照字面意义上理解，字典攻击就是遍历整个字典，对生成的字符串进行散列，并且把它和口令列表进行比较。

如果我们简单地把口令进行散列，如果两个或者多个用户选择了相同的口令，那么这将很快被暴露出来。更糟的是攻击者可以预计算一个把口令和散列组合起来的散列列表（单词hash

的另外一种用法）。整个“攻击”只需要在受害者的机器上花几秒钟的时间就可以完成。

2. 盐渍（Salts）

最常见而且也令人满意的解决办法（不用对用户进行培训）是给口令加“盐渍”。盐化的意思是我们给用户的散列“添加香料”。从密码学方面来看，它的意思是我们给散列的字符串加一些随机的数字。那些随机的数字，盐渍，将和用户所给定的证书证据列表联系起来。如果一个用户移动到另外一个系统，他会有一个和前面完全不同且没有任何关系的盐渍。盐渍能够防止攻击者所做的事情是预计算一个字典列表。攻击者只有在知道了用户的盐渍之后才能计算它，而且不得不对他选择攻击的每个用户都重新计算一次。

3. 盐渍的大小

既然我们正在对口令进行盐化，那么盐渍的大小应该是多少？特别地，它不需要太大。对于给定的证书列表中所有的用户，它应该是惟一的。一种安全的原则是使用不小于8个字节，不大于16个字节的盐渍。即使8个字节有些少，但这样它就不会过于影响性能（从存储空间或者计算时间上来说），这是一个很好的下限。

从技术角度来看，你至少需要将要保存的证书数量的平方个盐渍。例如，如果你的系统打算容纳1000个用户，那么你需要一个20位的盐渍。这是由于生日悖论。

我们的8个字节的建议允许你能在列表中存储多于40亿个的证书。

4. 重新散列（Rehash）

另外一种常用的技巧是不直接使用散列输出，而是把散列算法重新应用到散列输出上某些次。例如：

proof:= hash(hash(hash(hash(...(hash(*salt* || *password*)))))...)

虽然不是很科学，但这也是一种让字典攻击变得更慢的方法。如果你应用散列算法，假如是1024次，那么可以使得穷举搜索1024次更为困难。在实际应用中，用户将不可能注意到。例如，在一个AMD Opteron处理器上，1024次的SHA-1的调用大概花费将近720 000个CPU时钟周期。以平均的2.2GHz的时钟频率来算，这大概为0.32毫秒。这种用于由PKCS #5以同样的原因而使用。

5. 在线的口令

在线口令检查和离线是不同的。这里我们没有优先级，攻击者可以在客户端和服务器端之间截获并修改包。

最重要的第一步是建立一个逆名安全会话（anonymous secure session）。在客户端和服务器端之间的SSL会话就是一个很好的例子。这使得口令检查非常类似于离线时的情况。例如IKE和SRP（Secure Remote Passwords：http://srp.stanford.edu/）这样的协议都既实现了口令认证，也实现了信道安全（参见第9章）。

在没有这些解决方案的情况下，最好对口令使用一种挑战－应答（challenge-response）机制。基本的挑战应答是通过让服务器发送一个随机的字符串给客户端，然后客户端必须产生口令的消息摘要并且挑战以通过测试。要总是使用随机的挑战来防止重放攻击，这是很重要的。这种办法仍然易受中途相遇（meet in the middle）攻击，而且也不是一种安全的解决方案。

6. 两因素认证（Two-Factor Authentication）

两因素认证是一种用户校验方法，它用于在认证处理中有多种（在这个例子中至少为两种）不同形式的证书的情况。

它的一种非常流行的实现是RSA SecurID 令牌环（RSA SecurID token）。它们是小的，钥匙链大小的计算机，带有6～8个数字LCD。计算机已经为一个给定的用户ID设置了密钥。每分钟，它在LCD上产生一个新的数，这个数只有令牌环和服务器能知道。这种设备的目的是，仅靠猜密码是不足以破坏系统的。

实质上，这种设备是在产生一个秘密（服务器是知道的）和时间的散列。服务器必须补偿网络的偏差（通过允许在前一分钟，当前一分钟和下一分钟的值），但这还是很容易开发得。

5.4.4 性能上的考虑

散列算法并不需要像分组密码那么多的表查找或者复杂的操作。这使得性能（或者空间）的实现是一个相当不错而且短期的工作。

在SHS规范中的所有3个（不同的）算法也有相同的性能上的调整。

1. 内联扩展（Inline Expansion）

扩展的值（W[ ]数组）不需要在压缩之前完全计算出来。在每种散列算法中，在任何一个时刻只需要16个值。这意味着我们可以通过只存储这16个并在需要的时候再计算新的16个值来节省内存。

使用这种技巧，在SHA-1中，这可以节省256个字节的内存；SHA-256可以节省192个字节；SHA-512可以节省512个字节。

2. 压缩拆分（Compression Unrolling）

所有的3种算法都使用了一种类似于移位寄存器的结构。在一个完全组合起来的循环中，需要我们手动地从一个字到另一个字来移动数据。但是，如果我们完全拆分这个循环，就可以执行重命名来避免移动。所有的3个算法都有一个轮计数器，它是状态中字数的倍数。这意味着压缩结束时字的位置和开始时字的位置是相同的。

在SHA-1中，我们可以把4组循环都拆分成5-fold（译者注：5个为一组，后同），或者全部的20-fold。根据平台的不同，20-fold情况的拆分所得到的性能对于5-fold的拆分可能是正的，也可能是负的。在大多数桌面平台上，它并没变得更快，或者要以足够大的界限才能表明它变得更快。

在SHA-256和SHA-512中，循环拆分可以用8-fold或者全部的64-fold（80）来继续处理下去。由于SHA-256和SHA-512比SHA-1稍微复杂一点，所以在拆分方面的好处也有所不同。在Opteron上，处理完全拆分的SHA-256比8-fold所得到的性能提升要好得多，而SHA-512只有在8-fold时才会更好。

在后一种散列中拆分也意味着把轮常量嵌入到代码中而不用执行表查找的操作的可能性。这在像ARM这样的平台上所得到的效果很小，因为它无法在指令流中嵌入32位（或者64位）的常量。

3. 零复制散列

另一种优化是对正在散列的数据进行零复制。这种优化只要直接从用户传入的数据中加载消息分组而不需要用内部缓冲。这种散列在只有很少或者没有缓存的平台上非常重要。这些情况中的数据通常在一个相对比较慢的数据总线上传输，而且会因为流量而竞争系统设备。

例如，如果一个32位的加载或者存储操作需要6个时钟周期，它是低能源嵌入式设备的典型平均值，那么存储一个消息分组将花费96个时钟周期。一个压缩可能只花费1000到2000个时钟周期，因此我们用实际上并不需要的操作则增加了比原来多4.5%～9%的时钟周期。

这种优化通常只增加少量的代码，而且很容易就提升了性能。

5.4.5 PKCS #5的例子

我们现在将要考虑第4章中的AES CTR的例子。读者也许会有点厌烦，因为在那一节中，有许多代码是没有的，而只有“somehow fill secretkey and IV...（以某种方法来填充秘密密钥和IV...）”这样的注释。现在我们就展示一种填上它的方法。

读者应该记住我们用的是一个虚拟的口令来让示例运行的。在实际应用中，你可以从用户那里得到口令，或者通过首先把控制台关上等方法。

我们的例子再次用到了LibTomCrypt算法库。这个算法库也提供了一个不错的且方便的PKCS #5函数，它在一个调用中从秘密和盐渍产生输出。

pkcs5ex.c:

```
#include <tomcrypt.h>

void dumpbuf(const unsigned char *buf,
                   unsigned long  len,
                   unsigned char *name)
{
   unsigned long i;
   printf("%20s[0...%3lu] = ",name, len-1);
   for (i = 0; i < len; i++)  {
      printf("%02x ", *buf++);
   }
   printf("\n");
}
```

这是一个方便的调试函数，它可以用来转储数组（dumping arrays）。在密码学协议中，它有助于在最终输出之前看到中间输出。尤其是在多步协议中，它可以使得我们在偏离测试向量的那个地方进行调试。但是它要求测试向量列出这些数据。

```
int main(void)
{
   symmetric_CTR ctr;
   unsigned char secretkey[16], IV[16], plaintext[32],
                 ciphertext[32], buf[32], salt[8];
   int           x;
   unsigned long buflen;
```

与CTR例子中的变量列表类似。注意我们现在定义了一个salt[ ]数组和一个buflen整型变量。

```
   /* setup LibTomCrypt */
```

```
    register_cipher(&aes_desc);
    register_hash(&sha256_desc);
```

现在我们已经在加密算法库中注册了SHA-256。这允许我们在各种函数（例如PKCS #5）中通过名称来使用SHA-256。LibTomCrypt的这种方法的部分好处是，许多函数不清楚它们实际使用的是哪种分组密码、散列或者其他函数。我们的PKCS #5示例能够很容易地使用SHA-1、SHA-256或者甚至是Whirlpool散列函数。

```
    /* somehow fill secretkey and IV ... */
       /* read a salt */
       rng_get_bytes(salt, 8, NULL);
```

在这个例子中，我们读取RNG而不是设置一个PRNG。因为我们只要读取8个字节，这不太可能在Linux或者BSD的设置中阻塞。在Windows中，这从不阻塞。

```
       /* invoke PKCS #5 on our password "passwd" */
       buflen = sizeof(buf);
       assert(pkcs_5_alg2("passwd", 6,
                          salt, 8,
                          1024, find_hash("sha256"),
                          buf, &buflen) == CRYPT_OK);
```

这个函数调用了PKCS #5。我们传入的是虚拟的密码“passwd”，而不是一个合理的由用户所输入的密码。请注意这只是一个例子并且不是在你的应用程序中使用的密码机制。

紧接着的下一行指定了我们的盐渍及其长度——在本例中为8个字节。然后是所需次数的迭代。我们取1024只是因为它是一个不错的复杂的轮数。

find_hash()函数调用对于不熟悉LibTomCrypt算法库的读者来说可能是新的。这个函数搜索已注册的散列函数的表以找到和提供的名称相符的条目。它返回一个整数，表示它在表中的索引。然后该函数（本例中为PKCS #5）就可以使用这个索引来调用相应的散列算法。

LibTomCrypt使用的这个表实际上是一个C“struct”类型的数组，它包含了指向函数的指针以及其他参数。指向的函数实现了给定的不太清楚的散列算法。这使得调用程序能够从根本上支持任何散列算法而不用事先设计好它。

这个函数的最后一行指定了在哪里存储它以及读取多少数据。LibTomCrypt使用一个“caller specified（调用者指定的）”大小的缓冲区。这意味着调用函数首先应当说出缓冲区的大小（在一个指向unsigned long的指针中），然后该函数才会用存储的字节数来修改它。

这在公钥和ASN.1函数调用中很有用，因为调用函数并不总是知道最终输出的大小，但确实是知道它们所传入的缓冲区的大小。

```
       /* copy out the key and IV */
       memcpy(secretkey, buf,    16);
       memcpy(IV,        buf+16, 16);
```

此时，buf[0...31]包含了从我们的密码和盐渍中衍生出来的32个伪随机字节。我们复制前16个字节作为密钥，后16个字节作为CTR模式的IV。

```
    /* start CTR mode */
    assert(
      ctr_start(find_cipher("aes"), IV, secretkey, 16, 0,
                CTR_COUNTER_BIG_ENDIAN, &ctr) == CRYPT_OK);
```

```

    /* create a plaintext */
    memset(plaintext, 0, sizeof(plaintext));
    strncpy(plaintext, "hello world how are you?",
            sizeof(plaintext));

    /* encrypt it */
    ctr_encrypt(plaintext, ciphertext, 32, &ctr);

    printf("We give out salt and ciphertext as the 'output'\n");
    dumpbuf(salt, 8, "salt");
    dumpbuf(ciphertext, 32, "ciphertext");

    /* reset the IV */
    ctr_setiv(IV, 16, &ctr);

    /* decrypt it */
    ctr_decrypt(ciphertext, buf, 32, &ctr);

    /* print it */
    for (x = 0; x < 32; x++) printf("%c", buf[x]);
    printf("\n");

    return EXIT_SUCCESS;
}
```

在LibTomCrypt已经安装好的前提下，这个例子可以用如下的命令来构建。

```
gcc pkcs5ex.c -ltomcrypt -o pkcs5ex
```

这个例子的输出将类似于下面。

```
We give out salt and ciphertext as the 'output'
                salt[0...  7] = 58 56 52 f6 9c 04 b5 72
          ciphertext[0... 31] = e2 3f be 1f 1a 0c f8 96 0c e5 50 04 c0 a8 f7 f0
c4 27 60 ff b5 be bb bc f4 dc 88 ec 0e 0a f4 e6
hello world how are you?
```

每次运行都要选择一个不同的盐渍，并且分别产生一个不同的密文。正如这个演示程序所说明的，我们只能得到盐渍以及能够进行解密的密文（前提是我们知道口令）。我们不需要发送IV字节，因为它们是从PKCS #5算法中衍生而来的。

5.5 常见问题

下面的常见问题，由本书的作者所回答，它们即可以用来测试你对本章所出现的概念的理解，也可以帮助你在现实生活中实现这些概念。如果希望作者解答你的问题，请浏览www.syngress.com/solutions，然后点击“Ask the Author”表单。

问：什么是散列函数？

答：散列函数是接受任意长度的位串作为输入，并产生一个叫做消息摘要的固定大小的位串作为输出。一个密码学上的散列函数的目标是实现一种映射，好像这个函数是一个随机函数。

问：什么是消息摘要？

答：消息摘要就是散列函数的输出。通常，它被解释为消息的代表。

问：单向和碰撞约束是什么意思？

答：一个函数是单向的意思是指通过给定的输出来确定其输入是一个很难解决的问题。在本章中就是给定一个消息摘要，找到其输入应该是困难的。一个理想的散列函数是单向的。碰撞约束是说找出一对惟一的输入，使得它们产生相同的消息摘要是一个困难的问题。有两种形式的碰撞约束。第一种叫做预映射碰撞约束（pre-image collision resistance），意思是指给定一个固定的消息，我们不能找到另外一个消息和它碰撞。第二种简单地称之为第二预映射碰撞约束，意思是指找到两个随机的消息，使得它们能够碰撞是一个困难的问题。

问：散列函数是用来做什么的？

答：散列函数形成了所谓的伪随机函数（PRF）。即，从输入到输出的映射和一个随机函数是不好分辨出来的。作为PRF，散列函数可以用于完整性校验的目的。与一个存档一起包含一个消息摘要是使用散列的最直接的方法。散列函数也可以用来创建消息认证码（参见第6章），例如HMAC。散列函数也可以用于RNG和PRNG设计中的熵收集，并且产生了PRNG设计中的实际输出。

问：有哪些标准？

答：目前，NIST只指定了SHA-1和SHA-2散列算法系列作为标准。也有其他广泛使用的散列函数（通常是不幸的），例如MD4和MD5，这两个现在认为都已经被破解了。欧洲的NESSIE处理提供了Whirlpool散列算法，它是和SHA-512竞争的。

问：我可以在哪里找到这些散列算法的实现？

答：LibTomCrypt目前支持所有NIST的标准散列（包括较新的SHA-224），以及NESSIE指定的Whirlpool散列。LibTomCrypt也支持旧的散列算法，例如RIPEMD、MD2、MD4，等等，但是一般警告用户避免使用它们，除非您想要实现一个旧的标准（例如NT散列）。OpenSSL支持SHA-1和RIPEMD，Crypto++支持各种散列算法，包括NIST标准。

问：这些散列算法的专利是怎样的？

答：SHA-0（最初的SHA）过去曾是NSA的专利，但已经发布给公众，可以用于任何目的。SHA-2系列和Whirlpool都是公开的并且可以免费用于各种目的。

问：我应该使用什么摘要长度？什么是生日悖论？

答：一般来说，你应该使用两倍的消息摘要长度来作为你所寻求的位强度目标。例如，如果你希望一个攻击者要花费不少于2^{128}的工作来破解你的密码，那么你应该使用一个至少产生256位的散列算法。这是生日悖论的结果，它表明给定消息摘要定义域的平方根大小的输出，人们就能够找到一个碰撞。例如，对于一个256位的消息摘要，有2^{256}种可能的输出。其平方根是2^{128}，并且给定来自该散列的2^{128}个输入、输出对，攻击者可以在这个集合的条目中找到一个碰撞的概率很大。

问：什么是MD强化？

答：MD（Message Digest，消息摘要）强化是一种用消息长度的编码来对消息进行填充的技术，它可以避免各种前缀和扩展攻击。

问：什么是密钥衍生？

答：密钥衍生是取一个共享的秘密密钥并且从它产生出各种秘密和公开的材料（material），

以使得一个通信会话更加安全的处理过程。例如，通信双方能够同意一个私钥，然后把它传给一个密钥衍生函数来产生用于加密、认证的密钥以及各种IV参数。密钥衍生比较偏好于直接使用共享的秘密，因为它需要共享很少的位，而且也会减轻密钥探索所造成的损害。例如，如果一个攻击者知道了你的认证密钥，他也不可能知道你的加密密钥。

问：什么是PKCS #5？

答：PKCS #5是RSA Security公钥密码标准，它研究基于口令的加密问题。尤其是修改过的算法PBEKDF2（也叫做PKCS #5 Alg2）接受一个秘密的盐渍，然后把它扩展成用户需要的任意长度。从一个单独的（更短的）共享秘密中衍生出会话密钥和IV是非常有用的。尽管这个标准是为了基于口令的密码算法而制定的，但它也可以用于随机生成的共享秘密，它典型地用于公钥协商算法。

第 6 章

消息认证码算法

本章解决方案：

- 什么是MAC函数
- MAC的目标
- 安全准则
- 标准
- CMAC算法
- HMAC算法
- 总结

☑ 总结
☑ 快速查找解决方案
☑ 常见问题

6.1 简介

消息认证码（Message Authentication Code，MAC）算法是许多在线协议中相当关键的一个组件。它们保证了交易的双方或多方之间消息的认证。正如MAC算法的重要性一样，它们经常在加密系统的设计中受到重视。

一个典型的错误是过分地注意消息的保密性而不管消息被修改（可能是传输错误或者恶意的攻击者所造成的）所蕴含的信息。

一种更为常见的错误是，人们并没有意识到他们需要MAC算法。许多刚接触这个领域的人们假设不确定消息的内容就意味着你无法改变它。这种逻辑是，“如果他们不知道我的消息包含什么，他们又怎么有可能引入一个有用的改变呢”？

这种逻辑的错误首先是假设。通常，攻击者能够大概知道你的消息中的内容，而且这种知识足够以一种有意义的方式来弄乱消息。为了阐述这一点，考虑一个非常简单的银行交易协议。你传入一个交易给银行以进行认证，银行返回一个位：0为拒绝，1为成功的交易。

如果传输是没有认证的并且你能够在通信线路上改变消息，那么你可以制造各种麻烦。你可以发送一个假的证书给店主，而银行会适时地拒绝这一证书，但是因为你知道这个消息将要被拒绝，所以你能够把银行返回的加密了的0改成1——只要把这个位的值改动就可以了。设计MAC算法就是用来阻止这些类型的攻击的。

MAC算法和对称算法的工作环境大致相同。它们是固定的算法，接受一个控制输入到输出［常称为标记（tag）］的映射的秘密密钥。但是，MAC算法并不是在一个固定输入大小的基础上执行映射的，从这方面来说，它们也有些像散列函数，这让初学者有些迷惑。

尽管MAC函数接受任意大的输入并产生一个固定大小的输出，但它们在安全性方面并不等同于散列函数。带有固定密钥的MAC函数通常不是安全的单向散列函数。类似地，单向散列函数也不是安全的MAC函数（除非在某些特殊的情况下）。

MAC函数的目标

MAC的目标是保证共享一个秘密密钥的双方（或多方）在通信时能够具有检测传输的消息是否被修改（十有八九被修改了）的能力。这可以阻止攻击者修改消息以得到那些在前面讨论过的不希望他们得到的结果。

MAC算法通过接受消息和秘密密钥来作为输入，并且产生一个固定大小的MAC标记来完成这种工作。消息和标记会传输给其他的成员，然后可以重新计算标记并把结果和传输过来的标记进行比较。如果它们是相符的，那么消息几乎可以确定是正确的。否则，消息是不正确的，而且应该被丢弃，或者放弃这个连接，因为它很可能已被篡改，这取决于当时的环境。

对于一个想要伪造消息的攻击者来说，他会被要求攻破MAC函数。这显然不是一件很容易的事。实际上，你是希望对消息秘密的保护正如破解密码一样困难。

通常由于效率的原因，协议会把长的消息分成几片小的，它们将独立的去认证。这会产生各种问题，例如重放攻击。在本章的结尾，我们将讨论当使用MAC算法时协议的设计标准。简单地说，仅仅丢出一个合理的带有密钥的MAC算法来认证一个消息流是不够的。协议同样重要。

6.2 安全准则

MAC算法的安全性目标不同于单向散列函数的目标。与试图保证消息的完整性不同的是，我们试图确定其真实性。这些是不同的目标，但它们共享着许多背景知识。在这两种情况中，我们都试图确定正确性，或者更切确地说是消息的纯度。它们概念的不同之处在于真实性的目标也试图确定消息的来源。

例如，如果我告诉你一个文件的SHA-1消息摘要是160位的字符串*X*，然后把文件给你，或者更好地是你自己得到拿到这个文件，如果计算出来的消息摘要和给你的相符的话，那么你就能够确定这个文件是原始的（未修改过的）。但是你不会知道是谁创建了这个文件，消息摘要是不会告诉你的。假如我们处于通信的中间，并且我们都有一个共享的秘密密钥*K*。如果我发送给你一个文件以及用密钥*K*生成的MAC标记，那么你就可以通过校验这个MAC标记来验证这个消息是否是来自我的这个信道。

MAC和散列函数的另一个不同之处在于它们的位安全性的概念。回忆一下第5章，生日攻击可以把散列函数的位安全强度降低到摘要大小的一半。例如，对于SHA-256只需要花费2^{128}的工作就可以找到碰撞。这大概是因为消息摘要可以离线计算，这允许攻击者能够在不需要先攻陷受害者的情况下就可以预计算出一个巨大的消息摘要字典。另一方面，MAC算法只能是在线的。如果没有访问到密钥，碰撞是不可能找到的（如果MAC确实是安全的），而且如果不能以某种方式愚弄受害者来生成标记的话，攻击者是不能随便计算出标记的。

所以通常认为生日攻击是不适用于MAC函数的。也就是说，如果一个MAC标记是k位长，那么应该需要2^k的工作才能找到某个值的一个碰撞。你会经常看到那些很大的截短MAC标记的

长度以利用MAC函数的这种性质的协议。

例如，IPSec可以使用96位的标记。这是一种安全的优化，因为这个位安全仍然需要非常高的2^{96}的工作才能产生一个伪造。

6.2.1 MAC密钥的寿命

MAC的安全性不仅只取决于标记的长度。给定一个单独的消息和它的标记，标记的长度只能确定构建一个伪造的概率。但是，随着秘密密钥被用来认证越来越多的消息，所以其利益（advantage）——即一个成功伪造的概率——就提高了。

大致来讲，例如，对于基于分组密码的MAC，当达到生日悖论的极限之后，伪造的概率是0.5。即对于AES来说，在2^{64}分组之后，攻击者有一个均等的机会来伪造一个消息（这仍然是512EB的数据，确实惊人的信息数量）。

由于这个原因，我们必须不能仅从标记长度方面来考虑其安全性，也要从伪造的概率上来考虑。它规定了MAC密钥寿命的上限。幸运地是，我们不需要一个非常低的概率来保持安全。例如，如果有2^{-40}的伪造概率，攻击者可能首先会猜测正确的标记（或者和标记相符的内容）。仅此就意味着MAC密钥的寿命更有可能是学术讨论的问题，而不是在实际应用系统中我们需要担心的问题。

即使我们不需要一个非常低的伪造概率，但这并不意味着我们应该截短标记。只有当你认证了越来越多的数据时伪造的概率才会增加。对于短的消息来说，攻击者实际上并不知道任何需要用来计算伪造数据的信息，而且也依赖于用于攻击MAC的向量的随机碰撞概率。

6.3 标准

为了帮助开发者在他们的产品中实现可交互操作的MAC函数，NIST已经制定了两种形式的MAC函数标准。第一种是基于散列的HMAC（FIPS 198），它描述了一种把一个单向碰撞约束散列转化为一个MAC函数的安全的方法。虽然HMAC一开始打算是和SHA-1一起使用的，但它也适合于使用其他的散列函数（最近的研究结果表明，碰撞约束对于NMAC的安全性并不是必需的，NMAC算法是衍生自HMAC的，更详细的信息请参考http://eprint.iacr.org/2006/043.pdf。但是，另外一篇文章（http://eprint.iacr.org/2006/187.pdf）则建议不管怎样散列算法都要表现得更加安全）。

NIST开发的第二种标准是CMAC（SP 800-38B）标准。让人感到奇怪的是，CMAC在NIST的网站上被列为“操作模式”而不是一个消息认证码。抛开这个差异不看，CMAC确实是用于消息认证的。不像HMAC，CMAC使用一个分组密码来执行MAC函数，而且在只能适用一种分组密码的空间有限的条件下是很理想的。

6.4 分组消息认证码

分组消息认证码（Cipher Message Authentication Code，CMAC，SP 800-38B）算法实际上是取自一个叫做OMAC的提议，它表示“单一密钥消息认证码（One-Key Message Authentication Code）”，而且它在历史上是基于三密钥的密码分组链接MAC。由NIST提出的最初的基于分组密码的MAC形式上称为CBC-MAC。

在CBC-MAC设计中，发送者简单地选择一个单独的密钥（与加密密钥不是很容易就能联

系起来的）并以CBC模式继续对数据进行加密。发送者丢弃所有的中间密文，但除了最后一个，它是MAC的标记。如果用于CBC-MAC的密钥和用于加密明文的密钥是不相同的（或不相关的），那么MAC才是安全的（如图6-1所示）。

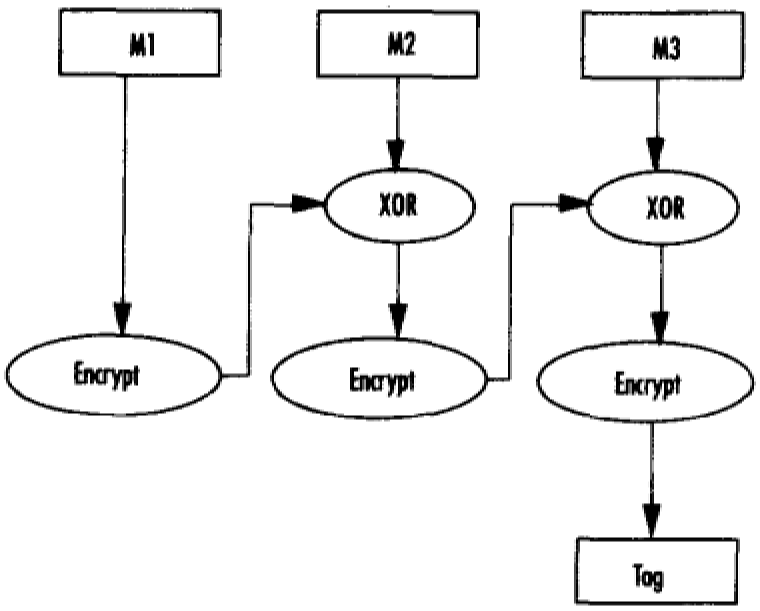


图6-1 CBC-MAC

即所有固定长度的消息都使用相同的密钥。当消息是具有不同长度的数据包时，这种机制就会变得不安全而且也有可能伪造，尤其是当消息不是分组密码的分组长度的偶数倍时。

为了解决这个问题，人们使用了XCBC的形式，它使用3个密钥。一个密钥用于密码算法以CBC-MAC模式来对数据进行加密。另外两个将和最一个消息分组进行异或，这取决于它是否是完全的。如果最后一个分组是完全的，那么将使用第二个密钥；否则，对这个分组进行填充并使用第三个密钥。

XCBC的问题是安全性的证据，至少是一开始的，要求3个完全不同独立的密钥。虽然提供一个密钥衍生函数是很简单的，例如PKCS #5，但通常提供这些密钥还是很困难的。

在XCBC模式之后是TMAC，它使用两个密钥。它和XCBC的工作流程类似，不同的是第三个密钥要线性的从第一个中衍生出来。它们用安全性来换取灵活性。用同样的方法，OMAC是TMAC的一个修定版本，它使用一个单一的密钥（如图6-2和图6-3所示）。

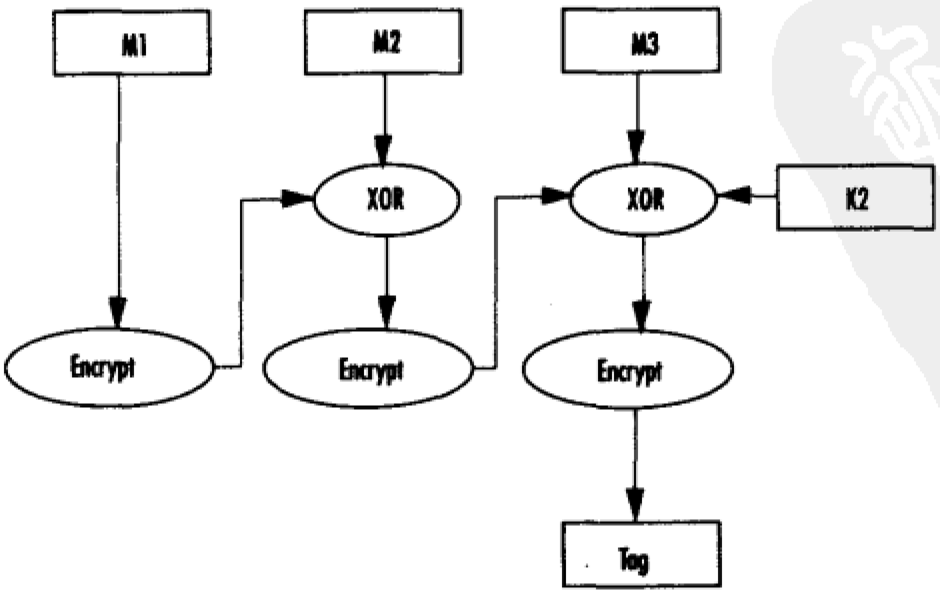


图6-2 OMAC整个分组消息

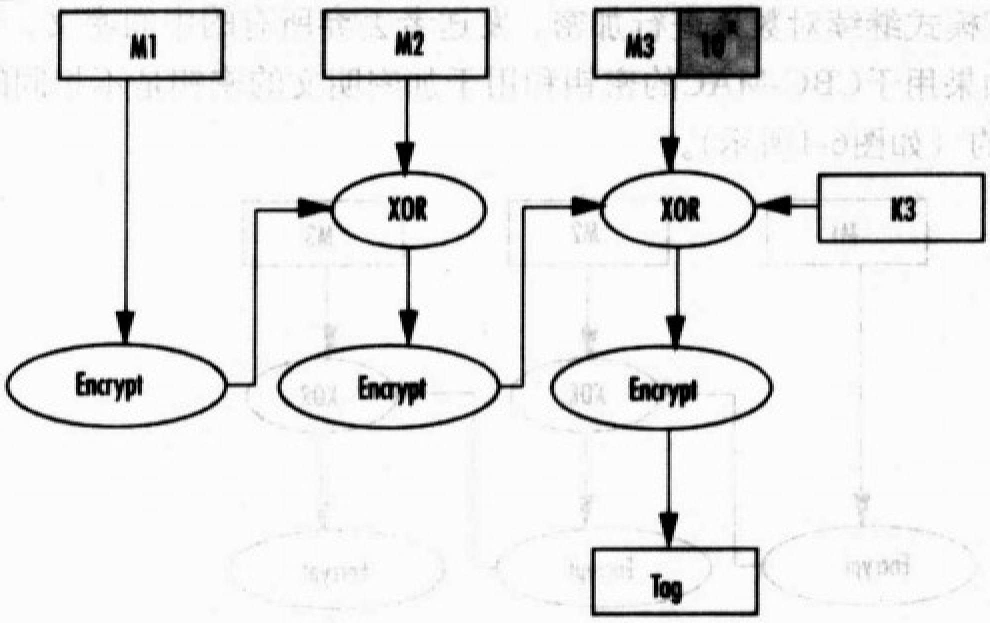


图6-3 OMAC部分分组消息

6.4.1 CMAC的安全性

为了让这些函数简单易用，他们让这些密钥互相依赖。如果攻击者知道了一个密钥，那么他就会知道其他的（或者在OMAC中，是所有的密钥）。我们所说的攻击者的利益（advantage）是指，在看到产生一定数量的MAC标记之后，他的伪造有可能成功的概率。

1. 令Adv^{MAC}代表一个MAC伪造的概率。
2. 令Adv^{PRP}代表把分组密码和随机置换区别开的概率。
3. 令t代表攻击者所花的时间。
4. 令q代表攻击者已看到的MAC标记的数目（以及相应的输入）。
5. 令n代表分组密码的分组大小（位）。
6. 令m代表每个认证的消息的（平均）分组数。

对于OMAC，攻击者的利益（大概）不多于：

$$Adv^{OMAC} < (mq)^2/2^{n-2} + Adv^{PRP}(t + O(mq), mq + 1)$$

假设mq非常小于$2^{n/2}$，那么AdvPRP()实际上为0。这样就剩下了等式的左边。这实际上给MAC算法定义了一个极限。假设我们使用AES（n=128），并且我们想要一个不大于2^{-96}的伪造概率。这意味着我们需要

$$2^{-96} > (mq)^2/2^{126}$$

如果我们对它进行化简，可以得到

$$2^{30} > (mq)^2$$

$$2^{15} > mq$$

它的意思是指使用相同的密钥，我们不能处理多于2^{15}个分组，同时保持伪造的概率低于2^{-96}。这种限制看起来有点严格，因为它意味着在不得不改变密钥之前，我们只能认证512KB的数据。回忆我们前面对MAC安全性的讨论，我们并不需要这种严格的要求。在攻击者被发现之前，他就失败了。假设我们使用上限2^{-40}来代替。这表示我们有如下的限制：

$2^{-40} > (mq)^2/2^{12}$

$2^{86} > (mq)^2$

$2^{43} > mq$

这意味着在改变密钥之前，我们可以认证2^{43}个分组（1024 TB）。看到所有传输的攻击者会有2^{-40}的伪造一个包的概率，可以相当安全地说这不会发生。当然，这并不意味着对于那个长度的流量就可以一直使用相同的密钥。

来自安全研究人员的忠告

在线与离线攻击

理解在线和离线攻击向量之间的区别是很重要的。为什么40位对于MAC来说就足够了，而对于分组密码的密钥却不行？

在一个MAC函数中，攻击是在线的。也就是说，攻击者不得不让受害者参加进来而且还要刺激他，让他给出一些消息。在传统的密码学文化中，我们把受害者称为"圣人"。因为所有的流量都会被认证，攻击者不能轻易地查询设备。但是，他可能会看到提供给MAC的已知数据。在任何事件中，对MAC的攻击是在线的。攻击者在被发现之前，只有一次伪造一个消息的机会。一个足够低的成功的概率，例如2^{-40}，意味着你可以安全地轻视这个问题。

在分组密码中，攻击是离线的。不需要受害者参与进来，攻击者就可以重复多次地执行一个给定的计算（例如用一个随机密钥来解密）。在这种情况下，一个40位密钥几乎不能提供任何持久的安全性。例如，一个AMD Opteron能够在将近2 000个处理器周期中对一个AES-128密钥进行测试。假设你使用了一个40位的密钥，其他的88个位都为0。一个2.2G Hz的Opteron将需要11.6天的时间来找到密钥。一个完全互补的AMD Opteron 885设置（4个处理器，总共8个核心，2.6GHz）能够以少于20 000美元的花费在大约1.23天中完成这个目标。

在自设计的硬件中，这会变得更糟。一个流行线AES-128引擎可以每个周期测试一个密钥，并且取决于FPGA以及预期的更大程度上的明文组合（例如ASCII），它可以接近100M Hz的速率。这将把搜索时间转变成3个小时。当然，这有点过分简化了，因为任何合理的快速密钥过滤方案会有许多错误的情况。它们需要一个辅助的（而且更慢）掩蔽处理过程。但是，它也可以并行工作而且因为比密钥会有更少的错误情况，所以测试不会变成瓶颈。

显然，在离线的情况下，位安全关系到很多方面。

6.4.2 CMAC的设计

CMAC是基于OMAC的设计，更为特别地是，它是基于OMAC1的设计。OMAC的设计者设计了两种非常相关的提议。OMAC1和OMAC2的不同之处仅在于怎样生成两个附加的密钥。在实际应用中，如果遵循CMAC标准的话，人们应该只使用OMAC1。

1. CMAC的初始化

CMAC在初始化过程中接受一个秘密密钥*K*作为输入。它使用这个密钥来生成两个附加的

密钥$K1$和$K2$。准确地说，CMAC使用一个在GF(2)[x]/v(x)域上的p(x) = x的乘法操作来完成这个密钥生成过程。幸运地是，有一种很容易的方法来解释这个过程（如图6-4所示）。

输入：
　　K: 秘密密钥
输出：
$K1$, $K2$：附加的CMAC密钥

1. $L = \text{Encrypt}_K(0)$
2. If MSB(L) = 0, then $K1 = L << 1$
   else $K1 = (L << 1)$ XOR R_b
3. If MSB($K1$) = 0, then $K2 = K1 << 1$
   else $K2 = (K1 << 1)$ XOR R_b
4. Return $K1$, $K2$

图6-4　CMAC初始化

这些值都是用big-endian格式来表示的，并且根据所用的分组密码不同，其操作都是64位或者128位的串。R_b的值取决于分组大小。对于128位的分组密码，它是0x87，64位的分组密码，它是0x1B。L的值是使用密钥K对所有0串的加密。

既然有了$K1$和$K2$，我们就可以继续MAC的处理过程了。紧记$K1$和$K2$必须保密是很重要的。把它们当成一个分组密码的密钥来看待。

2. CMAC的处理

从CMAC的描述来看，它只对你预先知道包的长度的情况下才有用。但是，由于惟一的差别发生在最后一个分组身上，可以把CMAC作为一个流MAC函数来实现，而不用事先知道数据的长度。对于长度为0的消息，CMAC把它们看成是不完全的分组（如图6-5所示）。

看起来给出C_i的值作为消息的密文是很吸引人的。但那会使针对CMAC的安全证据变得无效。你将不得不使用一个不同的（不相关的）密钥来加密你的明文以保持针对CMAC的安全证据。

3. CMAC的实现

为了能够使用第4章中的128位密钥的AES程序，我们对CMAC的实现进行了很好的设计。CMAC并不局限于这种情况，但为了更好地演示MAC，我们决定把它简化一下。LibTomCrypt中的CMAC程序（在OMAC目录里）演示了怎样去写一个非常灵活的能接受任何64位或者128位分组密码的CMAC程序。

cmac.c:
```
/* poor linker for AES code */
#include "aes_large_mod.c"
```

我们把AES代码复制到第6章的目录中。我们暂时想要保持代码的简洁，因此最后只是在应用程序中简单地直接包含了AES代码。

输入：

K: 秘密密钥

$K1, K2$: 附加的CMAC密钥

M: 消息

L: 消息的位数

$Tlen$: MAC标记所要求的长度

w: 每个分组的位数

输出：

T: 标记

1. If $L = 0$, let $n = 1$, else $n = \text{ceil}(L/w)$
2. 令 $M_1, M_2, M_3, \ldots, M_n$ 表示消息的分组
3. If $L > 0$ and $L \bmod w = 0$ then
   1. $M_n := M_n$ XOR $K1$
4. if $L = 0$ or $L \bmod w > 0$ then
   1. 追加一个‘1’位，然后是足够多的‘0’位以填满w位
   2. $M_n := M_n$ XOR $K2$
5. $C_0 = 0$
6. for i from 1 to n do
   1. $C_i = \text{Encrypt}_K(C_{i-1} \text{ XOR } M_i)$
7. $T = \text{MSB}_{Tlen}(C_n)$
8. Return T

图6-5 CMAC的处理

显然，在实际应用中，最好的实践是写一个AES头文件并把这两个文件恰当地链接在一起。

```
typedef struct {
   unsigned char L[2][16],
                  C[16];
   ulong32        AESkey[15*4];
   unsigned       buflen;
   int            first;
} cmac_state;
```

这是我们的CMAC状态函数。我们的实现将把CMAC消息作为一个数据流来处理，而不是一个固定大小的分组。数组L保存$K1$和$K2$两个密钥，我们是在cmac_init()函数中计算它们的。数组C保存CBC链接分组。我们通过把消息和数组C一行异或来把消息放到数组C中。整数*buflen*计算将要通过分组密码发送的字节数。

```
void cmac_init(const unsigned char *key, cmac_state *cmac)
{
   int i, m;
```

这个函数初始化我们的CMAC状态。它被很好地编码，以便使用128位的AES密钥。

```
    /* schedule the key */
    ScheduleKey(key, 16, cmac->AESkey);
```

首先，我们把输入的密钥调度到CMAC状态中的数组里。这使得我们能够在剩下的算法中按需要调用这个分组密码。

```
    /* encrypt 0 byte string */
    for (i = 0; i < 16; i++) {
       cmac->L[0][i] = 0;
    }
    AesEncrypt(cmac->L[0], cmac->L[0], cmac->AESkey, 10);
```

此时，我们的L[0]数组（等于$K1$）包含了0字节串的加密结果。我们将用x来乘上它，然后计算$K1$的最终值。

```
    /* now compute K1 and K2 */
       /* multiply K1 by x */
       m = cmac->L[0][0] & 0x80 ? 1 : 0;

       /* shift */
       for (i = 0; i < 15; i++) {
          cmac->L[0][i] = ((cmac->L[0][i] << 1) |
                           (cmac->L[0][i+1] >> 7)) & 255;
       }
       cmac->L[0][15] = (cmac->L[0][15] << 1) ^ (m ? 0x87 : 0);
```

我们首先取L[0]的MSB（赋给m），然后左移位。这个移位等于乘上x。最后一个字节在它自身上进行移位并且如果其MSB不为0的话，就用0x87和其异或。

```
       /* multiple K2 by x */
       for (i = 0; i < 16; i++) {
          cmac->L[1][i] = cmac->L[0][i];
       }
       m = cmac->L[1][0] & 0x80 ? 1 : 0;

       /* shift */
       for (i = 0; i < 15; i++) {
          cmac->L[1][i] = ((cmac->L[1][i] << 1) |
                           (cmac->L[1][i+1] >> 7)) & 255;
       }
       cmac->L[1][15] = (cmac->L[1][15] << 1) ^ (m ? 0x87 : 0);
```

这段代码是把L[0]($K1$)复制到L[1]($K2$)中，并再次乘上x。此时，我们就得到了CMAC用来处理消息的两个附加的密钥。

```
    /* setup buffer */
    cmac->buflen = 0;
```

```
        cmac->first   = 1;

        /* CBC buffer */
        for (i = 0; i < 16; i++) {
            cmac->C[i] = 0;
        }
    }
```

最后的一点是，代码初始化了缓冲区以及CBC链接变量。我们现在可以开始用CMAC处理消息了。

```
void cmac_process(const unsigned char *in, unsigned inlen,
                                     cmac_state *cmac)
{
```

我们的“process（处理）”函数非常类似于在散列算法的实现中看到的处理函数。它允许调用函数发送一个任意长度的消息来让这个算法进行处理。

```
    while (inlen--) {
       cmac->first = 0;
```

这是把第一个分组的标志关闭掉，以告诉CMAC函数我们在函数中至少已处理了一个字节。

```
       /* we have 16 bytes, encrypt the buffer */
       if (cmac->buflen == 16) {
          AesEncrypt(cmac->C, cmac->C, cmac->AESkey, 10);
          cmac->buflen = 0;
       }
```

如果我们已经填满了CBC链接分组，那么我们必须对它进行加密并且把计数器清零。我们必须对处理的每个16个字节都这样做，因为我们假设使用的是AES，它的分组大小是16个字节。

```
       /* xor in next byte */
       cmac->C[cmac->buflen++] ^= *in++;
    }
}
```

最后一个语句把消息中的一个字节和CBC链接分组进行异或。注意，在我们添加下一个字节之前，我们是怎样检查一个完整的分组的。这么做的原因在下一个函数中将变得更为明显。

这个循环在32位以及64位的平台上可以通过对输入消息和CBC链接分组的更大的字进行异或来优化。例如，在32位的平台上，我们可以向下面一样进行。

```
if (cmac->buflen == 0) {
   while (inlen >= 16) {
       *((ulong32*)&cmac->C[0])  ^= *((ulong32*)&in[0]);
       *((ulong32*)&cmac->C[4])  ^= *((ulong32*)&in[4]);
       *((ulong32*)&cmac->C[8])  ^= *((ulong32*)&in[8]);
       *((ulong32*)&cmac->C[12])  ^= *((ulong32*)&in[12]);
       if (inlen > 16) AesEncrypt(cmac->C, cmac->C, cmac->AESKey, 10);
```

```
        inlen -= 16;
        in    += 16;
    }
}
```

这个循环一次对所有的32位字进行异或，并且由于性能的原因，假设输入缓冲区是以32位来对齐的。注意到它的endian是不确定的，而且只取决于4个unsigned char映射到一个单独的ulong32。也就是说，这段代码并不是完全可移植的，但可以在大部分的平台上工作。注意，只有当CMAC缓冲区为空时，我们才会处理，而且只有当还剩下多于16个字节时，我们才会加密。

LibTomCrypt算法库使用了类似的技巧，这在64位平台上效果很好。算法库中的OMAC程序提供了另一种优化CMAC的例子。

提示 基于x86的平台打算创造“懒鬼”开发者。CISC指令集使得写相当高效的程序更加有效，尤其是具有以典型的类似于RISC的指令来使用内存操作数的能力——而在一个真正的RISC平台上，在对数据进行任何操作（例如加法）之前，你必须先加载它。

x86平台的另一个特性是允许未对齐。它们在性能上是半理想的，因为处理器必须使用成倍的内存指令来完善这种请求。但是，处理器仍然允许它的存在。

在其他平台上，例如MIPS和ARM，字的内存操作必须总是字对齐的。尤其是在ARM平台上，如果没有手动地模拟它们的话，你实际上不能执行未对齐的内存操作，因为处理器会把地址位清零。

如果C应用程序试图把一个指针转变为另外一种类型的话，这将会产生一些问题。正如在我们的例子中，我们把一个*unsigned char*指针转化为一个*ulong*32指针。这在x86平台上效果很好，但只有当指针是32位对齐的时候才能在ARM以及MIPS上工作。C编码器不能在编译时检测到这个错误，并且用户也只能在运行时环境下被告知有一个错误。

```
void cmac_done(   cmac_state *cmac,
                 unsigned char *tag, unsigned taglen)
{
   unsigned i;
```

这个函数结束CMAC并且输出MAC标记值。

```
   /* do we have a partial block? */
   if (cmac->first || cmac->buflen & 15) {
      /* yes, append the 0x80 byte */
      cmac->C[cmac->buflen++] ^= 0x80;

      /* xor K2 */
      for (i = 0; i < 16; i++) {
         cmac->C[i] ^= cmac->L[1][i];
      }
```

如果消息的长度为0或者是一个不完全的分组，那么首先追加一个值为1的位，然后是足够多的值为0的位。因为我们是基于字节的，填充就是字节0x80，然后是字节0，最后把*K*2和分组

进行异或。

```
    } else {
        /* no, xor K1 */
        for (i = 0; i < 16; i++) {
            cmac->C[i] ^= cmac->L[0][i];
        }
    }
```

否则，如果我们有一个完全的分组，那么就用$K1$和分组进行异或。

```
    /* encrypt pad */
    AesEncrypt(cmac->C, cmac->C, cmac->AESkey, 10);
```

我们对CBC的链接分组进行最后一次加密。这次加密的密文就是MAC标记。所有剩下的工作就是把它截短以满足调用函数的要求。

```
    /* copy tag */
    for (i = 0; i < 16 && i < taglen; i++) {
        tag[i] = cmac->C[i];
    }
}

void cmac_memory(const unsigned char *key,
                  const unsigned char *in, unsigned inlen,
                         unsigned char *tag, unsigned taglen)
{
    cmac_state cmac;
    cmac_init(key, &cmac);
    cmac_process(in, inlen, &cmac);
    cmac_done(&cmac, tag, taglen);
}
```

这个简单的函数允许调用者用一个单独的函数就能计算出一个消息的CMAC标记。使用起来很方便。

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    static const struct {
        int keylen, msglen;
        unsigned char key[16], msg[64], tag[16];
    } tests[] = {
    { 16, 0,
      { 0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6,
        0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c },
```

```
            { 0x00 },
            { 0xbb, 0x1d, 0x69, 0x29, 0xe9, 0x59, 0x37, 0x28,
              0x7f, 0xa3, 0x7d, 0x12, 0x9b, 0x75, 0x67, 0x46 }
        },
        { 16, 16,
            { 0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6,
              0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c },
            { 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96,
              0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a },
            { 0x07, 0x0a, 0x16, 0xb4, 0x6b, 0x4d, 0x41, 0x44,
              0xf7, 0x9b, 0xdd, 0x9d, 0xd0, 0x4a, 0x28, 0x7c }
        },
        { 16, 40,
            { 0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6,
              0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c },
            { 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96,
              0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
              0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c,
              0x9e, 0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51,
              0x30, 0xc8, 0x1c, 0x46, 0xa3, 0x5c, 0xe4, 0x11 },
            { 0xdf, 0xa6, 0x67, 0x47, 0xde, 0x9a, 0xe6, 0x30,
              0x30, 0xca, 0x32, 0x61, 0x14, 0x97, 0xc8, 0x27 }
        },
        { 16, 64,
            { 0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6,
              0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c },
            { 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96,
              0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
              0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c,
              0x9e, 0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51,
              0x30, 0xc8, 0x1c, 0x46, 0xa3, 0x5c, 0xe4, 0x11,
              0xe5, 0xfb, 0xc1, 0x19, 0x1a, 0x0a, 0x52, 0xef,
              0xf6, 0x9f, 0x24, 0x45, 0xdf, 0x4f, 0x9b, 0x17,
              0xad, 0x2b, 0x41, 0x7b, 0xe6, 0x6c, 0x37, 0x10 },
            { 0x51, 0xf0, 0xbe, 0xbf, 0x7e, 0x3b, 0x9d, 0x92,
              0xfc, 0x49, 0x74, 0x17, 0x79, 0x36, 0x3c, 0xfe }
        }
        };
```

这些数组是使用AES-128的CMAC的标准测试向量。一个实现至少要符合这些向量以声明是遵循CMAC AES-128的。

```
        unsigned char tag[16];
        int i;

```

```
    for (i = 0; i < 4; i++) {
        cmac_memory(tests[i].key, tests[i].msg,
                    tests[i].msglen, tag, 16);
        if (memcmp(tag, tests[i].tag, 16)) {
            printf("CMAC test %d failed\n", i);
            return -1;
        }
    }
    printf("CMAC passed\n");
    return 0;
}
```

这个演示程序计算了测试消息的CMAC标记并进行比较。记住这个测试程序只使用了AES-128而不是整个AES系列。尽管如此，但一般来说，如果能符合AES-128 CMAC测试向量，应该也能符合AES-192和AES-256的测试向量。

4. CMAC的性能

总的来说，CMAC的性能取决于使用的分组密码。通过对处理函数的反馈优化（用字为单位进行异或而不是字节），开销可以达到最小。

不幸地是，CMAC所用的是CBC模式而且是不能并行的。这意味着在硬件环境中，最好的性能是通过最快的AES实现，而不是通过许多并行的实例来达到的。

6.5 散列消息认证码

散列消息认证码（Hash Message Authentication Code，HMAC）标准（FIPS 198）使用密码学上的单向散列函数并将它转化为消息认证码算法。还记得之前我们曾说过散列不是认证函数吗？这一节将告诉你怎样把你喜欢的散列转变为一个MAC。

所有的HMAC设计衍生自一个叫做NMAC的提议，它用可证明的安全界限把任意的伪随机函数（PRF）转化为一个MAC函数。特别地，其关注的是用散列函数来做PRF。NMAC是基于在消息前面加一个密钥，然后对这个连接进行散列的概念。例如，

$tag = \text{hash}(key \| message)$

但是，回忆一下我们在第5章中说过这种结构是不安全的。尤其是攻击者能够通过把标记作为散列的初始状态来扩充消息。问题是攻击者知道散列的消息以产生标记。如果我们能够以某种方式把它隐藏起来，攻击者就不能产生一个有效的标记。事实上，我们有

$tag = \text{hash}(key \| \text{PRF}(message))$

现在有一个想要通过使用标记作为初始散列状态的攻击者，而且PRF()映射的结果将是不可预测的。在这种配置中，攻击者不能再使用标记来作为初始状态了。现在惟一的问题是怎样创建一个PRF？可以证明最初的构造是一个相当好的PRF。也就是说，散列函数的定义就是伪随机函数，把它们的输入映射到很难预测的输出中。其输出也是很难逆过来的（即散列是单向的）。通过预先把秘密数据放到消息前面来密钥化散列函数，这样可以定义一个带有密钥的PRF。

完整的结构为：

tag = hash(*key*1 ‖ hash(*key*2 ‖ *message*))

注意NMAC需要两个密钥，一个用于内散列，另一个用于外散列。虽然这个结构简单，但它效率很低，因为它需要两个独立的密钥。

HMAC的结构基于NMAC，不同的是其中的两个密钥是线性相关的。HMAC的贡献是，证明了只用一个单独的密钥结构也是安全的。它要求散列函数是一个安全的PRF，虽然它不需要是抗碰撞的（针对NMAC和HMAC的新证明：Security without Collision Resistant: http://eprint.iacr.org/2006/043.pdf），但它必须是抗差分分析的（On The Security of HMAC and NMAC Based on HAVAL, MD4, MD5, SHA-0, and SHA-1：http://eprint.iacr.org/2006/187.pdf）。

6.5.1 HMAC的设计

既然我们已经大致地了解了什么是HMAC，那么我们就可以研究FIPS 198标准的特殊之处。HMAC，类似于CMAC，并不特定于某种给定的算法。虽然HMAC最初打算是用于SHA-1的，但它也可以安全地用于其他安全散列函数，例如SHA-256和SHA-512。

HMAC从一个单独的秘密密钥衍生出两个密钥，这是它通过和两个常量进行异或完成的。首先，在我们这样做之前，需要确定这个密钥是散列压缩分组的大小。例如，SHA-1和SHA-256压缩64个字节的分组，而SHA-512压缩128个字节的分组。

如果这个秘密密钥比压缩的分组大小要大，这个标准要求首先将这个密钥进行散列，然后把散列的输出作为秘密密钥。秘密密钥也需要用0字节来填充以保证是压缩分组输入的大小。

填充后的结果被复制两次。第一次复制的所有字节都要和0x36进行异或，这是外部的密钥。另外一次复制的所有字节都要和0x5C进行异或，这是内部的密钥。为简便起见，我们把0x36字节的串叫做*opad*，把0x5C字节的串叫做*ipad*。

你也许想知道为什么密钥要填充为压缩分组的大小。实际上，这通过让初始散列状态依赖于密钥来把一个散列转变为一个带有密钥的散列。从密码学的观点来看，HMAC等价于根据秘密密钥来随机地提取散列的初始状态（如图6-6和图6-7所示）。

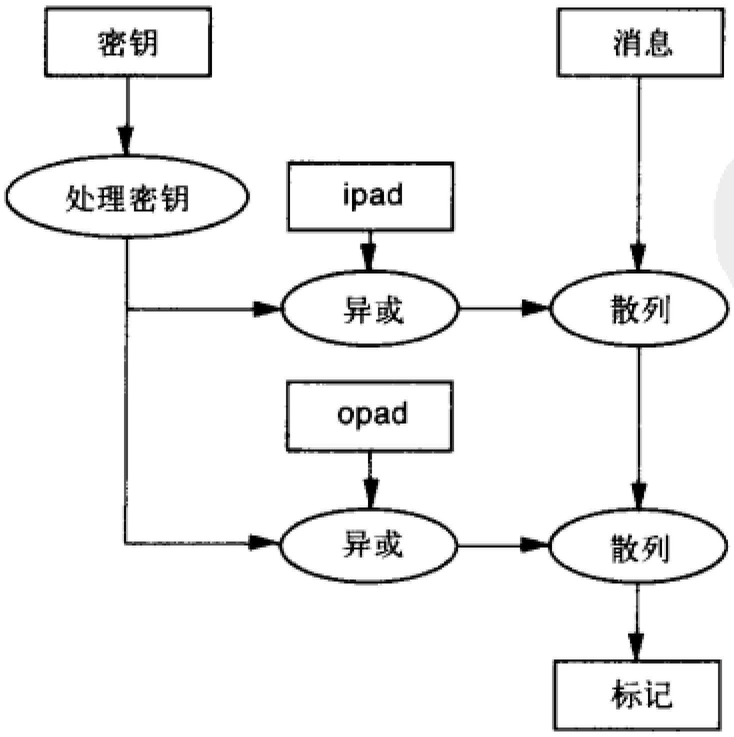


图6-6 HMAC分组示意图

输入：

K: 秘密密钥

message: 用来确定MAC是做什么的消息

w: 压缩分组大小

Tlen: 要求的MAC标记长度

输出：

Tag: MAC标记

1. if length(*K*) > *w* then
    1. *K* = hash(*K*)
2. 用0来填充K直到它为w字节长
3. *Tag* = hash((*opad* XOR *K*) || hash((*ipad* XOR *K*) || *message*))
4. 通过只保留前Tlen个字节把标记截短为Tlen字节
5. 返回标记

图6-7 HMAC算法

正如我们所看到的，HMAC是一个很容易描述的算法。通过一个方便的散列函数的实现，就可以很简单地实现HMAC。我们注意到，由于消息仅在内散列中进行散列，实际上可以动态地用HMAC来处理一个数据流而不用一次需要全部的消息（前提是散列函数并不是一次就需要全部的消息）。

由于这个实现用一个散列来作为PRF，并且大多数的散列都不能并行地处理消息分组，所以从性能方面来看，这会成为算法的一个关键路径。在计算内散列的同时也可以计算（opad XOR K）的散列，但这只能节省很少的时间而且也需要两个并行的散列实例。对于仅是几个消息分组长度的消息来说，这种优化不太值得。

6.5.2 HMAC的实现

HMAC的实现和SHA-1散列函数的实现是紧密相关的。类似于CMAC，我们决定简化这个实现以保证这个实现是容易被读者理解的。

hmac.c:

```
/* poor linker */
#include "sha1.c"
```

直接包含了SHA-1源代码以提供我们的散列函数。理想情况下，我们会包含一个合适的头文件并且和SHA-1同时链接。但是，此时我们只想证明HMAC是有效的。

```
typedef struct {
   sha1_state     hash;
   unsigned char K[64];
} hmac_state;
```

这是我们的HMAC状态。它相当简单，因为所有的数据缓冲区都是由SHA-1函数在内部处

理的。我们保存外部密钥的一个副本以允许HMAC实现能够应用外散列。注意我们将不得不改变K的大小以适应这个散列。例如，如果是SHA-512，那么它将会是128个字节长。我们也会对下面的函数做同样的改变。

```
void hmac_init(const unsigned char *key,
                                unsigned  keylen,
                             hmac_state *hmac)
{
    unsigned char K[64];
    unsigned i;
```

这个函数通过处理密钥以及开始内散列来初始化HMAC的状态。

```
    /* if keylen > 64 hash it */
    if (keylen > 64) {
        sha1_memory(key, keylen, K);
        i = 20;
    } else {
        /* copy key */
        for (i = 0; i < keylen; i++) {
             K[i] = key[i];
        }
    }
```

如果秘密密钥比64个字节（SHA-1的压缩分组大小）要大，我们使用辅助的SHA-1函数对它进行散列。如果它没有的话，那么就把它复制到局部数组K中。在这两种情况中，此时i会表示数组中已经填入的字节数。它会在下一个循环中用来填充密钥。

```
    /* pad with zeros */
    for (; i < 64; i++) {
         K[i] = 0x00;
    }
```

这里用0字节来填充密钥，因此它会是64个字节的长度。

```
    /* copy key to structure, this is out outer key */
    for (i = 0; i < 64; i++) {
        hmac->K[i] = K[i] ^ 0x5C;
    }

    /* XOR inner key with 0x36 */
    for (i = 0; i < 64; i++) {
        K[i] ^= 0x36;
    }
```

第一个循环创建外部密钥并把它保存在HMAC状态中。第二个循环创建内部密钥把它保存在局部变量中。我们只需要它很短的时间，因此没有理由要把它复制到HMAC状态中。

```
    /* start hash */
```

```
    sha1_init(&hmac->hash);

    /* hash key */
    sha1_process(&hmac->hash, K, 64);

    /* wipe key */
    for (i = 0; i < 64; i++) {
       K[i] = 0x00;
    }
}
```

此时，我们已经初始化了HMAC的状态。我们可以用如下的函数来处理将要被认证的数据。

```
void hmac_process(const unsigned char *in,
                                    unsigned  inlen,
                                 hmac_state *hmac)
{
    sha1_process(&hmac->hash, in, inlen);
}
```

这个函数处理我们想要认证的数据。花一些时间来鉴赏HMAC的巨大的复杂性。好了？这是HMAC是一个好的标准的原因之一。它实现起来是如此的简单。

```
void hmac_done(    hmac_state *hmac,
                unsigned char *tag,
                       unsigned  taglen)
{
```

这个函数结束HMAC并输出标记。

```
     unsigned char T[20];
     unsigned       i;

```

数组*T*存储了来自散列函数的消息摘要。你将不得不对它进行调整以符合散列函数的输出大小。

```
     /* terminate inner hash */
     sha1_done(&hmac->hash, T);

     /* start outer hash */
     sha1_init(&hmac->hash);

     /* hash the outer key */
     sha1_process(&hmac->hash, hmac->K, 64);

     /* hash the inner hash */
     sha1_process(&hmac->hash, T, 20);
```

```

    /* get the output (tag) */
    sha1_done(&hmac->hash, T);

    /* copy out */
    for (i = 0; i < 20 && i < taglen; i++) {
       tag[i] = T[i];
    }
}
```

此时，我们已经具备了用HMAC来处理数据的所有先决条件。我们可以借助于我们的散列实现到达这里。

```
void hmac_memory(const unsigned char *key,
                                    unsigned  keylen,
                       const unsigned char *in, unsigned inlen,
                             unsigned char *tag, unsigned taglen)
{
   hmac_state hmac;
   hmac_init(key, keylen, &hmac);
   hmac_process(in, inlen, &hmac);
   hmac_done(&hmac, tag, taglen);
}
```

正如CMAC中的一样，我们提供了一个用起来很简单的HMAC函数，它可以在一个单独的函数调用中产生一个标记。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
     static const struct {
         unsigned char key[128];
         unsigned long keylen;
         unsigned char data[128];
         unsigned long datalen;
         unsigned char tag[20];
     } tests[] = {
     {
       {0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c,
        0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c,
        0x0c, 0x0c, 0x0c, 0x0c}, 20,
       "Test With Truncation", 20,
       {0x4c, 0x1a, 0x03, 0x42, 0x4b, 0x55, 0xe0, 0x7f,
```

```
            0xe7, 0xf2, 0x7b, 0xe1, 0xd5, 0x8b, 0xb9, 0x32,
            0x4a, 0x9a, 0x5a, 0x04} },
      {
          {0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
            0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
            0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
            0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
            0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
            0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
            0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
            0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
            0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
            0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa}, 80,
          "Test Using Larger Than Block-Size Key - "
          "Hash Key First", 54,
          {0xaa, 0x4a, 0xe5, 0xe1, 0x52, 0x72, 0xd0, 0x0e,
            0x95, 0x70, 0x56, 0x37, 0xce, 0x8a, 0x3b, 0x55,
            0xed, 0x40, 0x21, 0x12} },
      {
          {0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
            0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
            0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
            0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
            0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
            0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
            0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
            0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
            0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
            0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa}, 80,
          "Test Using Larger Than Block-Size Key and Larger "
          "Than One Block-Size Data", 73,
          {0xe8, 0xe9, 0x9d, 0x0f, 0x45, 0x23, 0x7d, 0x78,
            0x6d, 0x6b, 0xba, 0xa7, 0x96, 0x5c, 0x78, 0x08,
            0xbb, 0xff, 0x1a, 0x91} } };
   unsigned char tag[20];
   unsigned i;

   for (i = 0; i < 3; i++) {
      hmac_memory(tests[i].key, tests[i].keylen,
                  tests[i].data, tests[i].datalen,
                  tag, 20);
      if (memcmp(tag, tests[i].tag, 20)) {
         printf("HMAC-SHA1 Test %u failed\n", i);
         return -1;
```

```
        }
    }
    printf("HMAC-SHA1 passed\n");
    return 0;
}
```

我们的实现中的测试向量来自于1997年出版的RFC 2002（HMAC RFC：www.faqs.org/rfcs/rfc2104.html，HMAC Test Cases：www.faqs.org/rfcs/rfc2202.html）。请求注解（Request for Comments，RFC）定义了HMAC算法最初的标准。我们使用这些测试向量是因为它们是在FIPS 198中所列出的测试向量之前公布的。严格来说，FIPS 198并不依赖于RFC 2104；也就是说，为了证明标准是遵循FIPS 198的，你必须传入FIPS 198的测试向量。

不幸的是，RFC 2104和FIPS 198定义了同样的一个算法。更让人感到奇怪的是，在NIST FIPS 198规范中，他们声称他们的标准是RFC 2104的"泛化（generalization）"。我们还没有注意到这两个标准之间有什么重大的区别。虽然HMAC起初是打算使用MD5和SHA-1的，但RFC并没有说它有这种限制。事实上，引自RFC，"这个文档定义了使用一般意义上的密码学散列函数（用H来表示）的HMAC"，我们能够看到这个标准的范围并没有限定为一个给定的散列函数。

6.6 总结

既然我们已经了解了两种标准的MAC函数，那么我们就可以开始研究怎样用一种安全的方式来使用MAC函数去完成有用的目标。我们将研究基本的任务，即MAC函数可以用来做什么，不可以用来做什么。接着，我们将把CMAC和HMAC做一个比较并给出在什么时候选择哪个的建议。

在此之后，我们将研究使用哪个MAC函数以及为什么我们想要您去使用它们。我们还会继续研究怎样安全地使用它们。把MAC函数简单地应用于你的数据还不够，为了系统的安全，你还必须在一个给定的环境中使用它。

6.6.1 MAC函数可以做哪些事

首先以及最重要的是，MAC函数设计用来提供在一个通信信道上成员之间的认证。如果一切都正常运行，所有的成员都可以互相发送和接收（以及校验）经过认证的数据——也就是说，防止攻击者修改消息的情形以很高的概率出现。

那么，在现实世界环境中，我们到底能用MAC来做什么呢？一些经典的例子包括Internet上的SSL和TLS连接。例如，HTTPS连接使用TLS或者SSL来加密信道以进行保密，并且对在各个方向上发送的数据应用MAC以确保其认证。SSH是另一种经典的例子。SSH（Secure Shell，安全Shell）是一种协议，它允许用户用类似于telnet的方式远程登录到其他机器上。但是，不像telnet，SSH使用加密算法来保证在两个方向上发送的数据的保密性和认证。最后，另外一个例子是GSM蜂窝标准。它使用一种称作COMP128的算法来认证网络上的用户。不像SSL和SSH，COMP128通过提供一个挑战（challenge），作为回应，用户必须提供正确的MAC标记来对用户进行认证。COMP128并不用于对传输数据的认证。

结果（Consequences）

假设在前面的协议中我们没有使用MAC认证函数，那么用户将会面临什么样的威胁？

在TLS和SSL的典型用例中，用户试图连接一台合法的主机并在这两点之间传输数据。这些数据通常由HTTPS流量组成，即HTML文档和表单回复。能够插入或者修改回复的结果取决于其应用。例如，如果你正在阅读电子邮件，一个攻击者可以从受害者重新发送前一个命令给服务器。这些命令可以是没有危害的，例如“阅读下一封电子邮件”，但也可以是破坏性的，例如“删除当前电子邮件”。网上银行也是TLS加密的另一个重要的用户。假设你发送了一个“付给接受者100美元”的HTML表单回复。一个攻击者可以修改这个消息并且改变美元数量，或者更简单一些，不断地重新发送这个命令直到耗尽银行账户。

在TLS和SSL域中的修改通常不太重要，因为攻击者是在修改密文，而不是明文。攻击者几乎不能用合法的方式做修改。例如，一个通过TLS连接到SMTP的用户必须发送合理的格式化的命令以让服务器执行工作。改变数据将可能会修改命令。

提示 在像TLS这样的协议中不使用MAC的一个结果是攻击者能修改密文。在加密模式中，例如CTR，密文的1位的改变只能导致明文的1位的改变。

CBC不是一个抗伪造的操作模式。一个知道一些明文和密文对并且能够控制所用的CBC IV的攻击者，能够伪造一个会被合理解密的消息。

如果你需要认证，而且你这么做的机率不错的话，CBC模式并不是达到这种目的的方式。

在SSH环境中，如果没有放置一个MAC，攻击者可以重新发送许多用户发送过的命令。这是因为SSH使用一个类-Nagle的协议来缓冲发送出的数据。例如，如果SSH为你的每次按键发送一个包，简单地输入几段文字它就会发送几KB的数据。类似于TCP/IP协议，SSH把将要发送的数据缓冲起来直到收到另一端的回复。用这种方式，这种协议就可以是自同步的（self-clocking），而且在使用一个最理想数量的带宽时会产生和网络允许一样的低延迟。

输入缓冲带来了某些问题。例如，如果你在shell提示符下敲出“rm -f *”以清空一个目录，很可能整个字符串（和新的行一起）可能会被作为一个数据包发送到SSH服务器。一个接收到这个包的攻击者就能够重放它，不管在会话中你当前在哪个目录下。

在GSM的COMP128协议中，如果MAC不存在的话（并且不使用替代品），一个GSM客户端就可以很容易地跳到一个蜂窝网络中说“I'm Me!”并且能够发送任何他想要发送的命令，例如拨出一个号码或者初始一个GPRS（数据）会话。

简而言之，如果没有一个MAC函数的话，攻击者能够修改数据包并且重放旧的包，而不会被马上以及可能检测到。一开始看起来修改包很困难，但是，在各种协议中如果攻击者具有某些明文的可能特征的知识，那么这是完全有可能的。人们也许认为修改会被完全捕捉到。例如，如果消息是文本，人工阅读就可以注意到。如果消息是一个可执行文件，处理器会在无效指令处抛出一个异常。但是，并不是所有的修改，甚至是随机的（在没有攻击者的情况下），都是那样令人注目的。

例如，对于内存是坏的机器上的很多情况——也就是说，内存没有正确地存储数值，或者

没有遵循时间规范——处理器收到的错误将在相当的时间内没有被注意到。有问题的内存的征兆可以像一个程序的终止那样简单，或者是文件和目录没有正确地显示。服务器通常使用纠错码（Error Correcting Code，ECC）内存，它可以和数据一起发送冗余信息，然后ECC数据就会用来动态地纠正错误。但是，即使使用ECC，也有可能存在无法纠正的内存失败的情况。

消息重放攻击会更加危险。在这种情形中，攻击者重新发送一个旧的有效的包，希望给受害者造成破坏。重放的包是没有被修改的，因此其解密是有效的，而且根据当前协议的环境，这可能会非常危险。

6.6.2 MAC函数不能用来做哪些事

MAC函数通常在加密应用中并没有过多的使用，至少没有像散列和分组密码那样。这有些让人感到悲痛，这并不是因为大多数的开发者在技术上对MAC函数的理解。事实上，大多数成熟的协议也一起考虑了认证的威胁。顺便提一下，那通常是一个很好的蛇油（译者注：蛇油(snake oil)，在网络安全中，它是指供应商夸大其词的宣称。19世纪中期，密码学家把这种供应商比作卖药的小贩，他们所鼓吹的加密算法就像街头卖药者鼓吹的神秘配方一样。）产品的例子。MAC函数通常不会被那些了解它们的人滥用，大多是因为MAC函数并不能像分组密码和散列函数那样成为有用的原型。虽然这样说，但我们还要简要地介绍一些基本的东西。

MAC函数并不打算像散列函数那样去使用。有时，人们会要求使用带有一个固定密钥的CMAC来创建一个散列函数。这种结构是不安全的，因为其“压缩”函数（分组密码）不再是一个PRP。没有任何证据可以证明这一点。存在一种安全的把一个分组密码转化为一个散列的方法：

1. H[0] = $Encrypt_0(0)$。
2. 用MD强化对消息进行填充并且分成几个分组M[1]，M[2]，…，M[n]。
3. i从1到n　H[i] = Encrypt H([i−1]) (m[i]X) COR M[i]。
4. 返回H[n]作为消息摘要。

在这种模式中，EncryptK(P)意思是用密钥K对明文P进行加密。初始值H[0]是为了让协议可以更简单地定义而对0串进行加密所选择的。这种散列算法不是任何NIST标准的一部分，尽管它已用于各种产品中。它是安全的，但前提是分组密码本身是抗差分攻击的，并且密钥调度是抗相关密钥攻击的。

MAC也不是用于RNG处理的，或者也不是用于转化为一个PRNG的。使用一个随机的秘密密钥，MAC函数对攻击者应该表现得像一个PRF一样。这意味着它可以作为一个PRNG来使用，这确实是真的。在实际应用中，如果你运行一个作为输入的计数器和使用输出作为随机数的话，一个合适的密钥化了的MAC可以构造出一个安全的PRNG。但是，这种结构很慢而且浪费资源。一个合适的密钥化了的工作于CTR模式的分组密码可以实现同样的目标，而且消费更少的资源。

6.6.3 CMAC与HMAC

把CMAC和HMAC进行比较，其衡量标准取决于你想要解决的问题以及你要用什么方法来解决它。如果你已经有了一个分组密码的话，CMAC典型地是一种不错的选择（用于保密）。

你也可以重用同样的代码（或者硬件）来实现认证。CMAC也是基于分组密码的，它虽然有很小的输入（和散列函数比较），但是能够以更短的次序返回一个输出。这降低了操作的延迟。对于短的消息，小至16或者32个字节的，带AES的CAMC经常是比HMAC更快并且延迟更低。

HMAC开始胜过CMAC的是对于更长的消息。散列函数和分组密码相比较而言，处理每个字节需要更少的时钟周期。散列也产生比分组密码更大的输出，这可以用于要求更大的MAC标记的情况。不幸地是，HMAC要求你已实现了一个散列函数，但是，从好的方面来看，围绕着散列实现的HMAC代码是很容易实现的。

从安全性的角度来看，前提是每个原型（即分组密码或者散列函数）自身都是安全的（即分别为一个PRP或者一个PRF），其MAC结构也可以是安全的。虽然HMAC比CMAC能够产生更大的MAC标记（通过使用一个更大的散列函数），但使用更大标记的安全利益并不是很明显。

底线是：如果你有足够的空间或者已经有一个散列，而且你的消息并不是非常小，那么就使用HMAC。如果你有一个分组密码（或者想要一个更小的“足迹”）而且正在处理更小的消息，那么就使用CMAC。最重要地是，不要重复造轮子；使用适合你的标准。

6.6.4 重放保护

简单地对你的消息应用MAC函数还不足以保证整个系统对于重放攻击是安全的。这种问题是由于对效率和必要性的需求而产生的。一个程序不是把一个消息作为一个长的数据块进行发送，而是把它分成更小的，更方便管理的一些片并且按顺序发送它们。流的应用，例如SSH，实际上要求使用数据包以一种有用的容量来发挥作用。

使用包的加密缺陷，即使是独立进行认证的包，是它们将作为一个整体来形成一个更大的消息。也就是说，那些各自独立的包不仅自己必须是正确的，而且在整个会话中包的顺序也必须正确。攻击者可能通过重放旧的包或者修改包的序号来利用这一点。如果系统没有办法来认证包的序号，它就可以把包作为有效的来接受并把这个重新排列解释为整个消息。

可以表明这是一个问题的经典的例子是购买编号。例如，一个客户想要购买一个公司的100股股票，因此他把消息打包为M=“我想买100股”。然后客户对这个消息进行加密并且对密文应用一个MAC。为什么这是不安全的？一个看到了密文和MAC标记的攻击者能够重新传送这对消息。服务器，它和重放攻击没有关系，将会读取这对消息，校验MAC并把密文解释为有效。现在攻击者可以通过想重新发送多少次购买编号就发送多少次来耗尽受害者的资金。

两种针对重放攻击的典型解决方法是时间戳（timestamp）和计数器。两种解决方案都必须包含额外的数据作为已认证消息的一部分。它们给包一个环境（上下文），这有助于接收者解释它们在数据流中出现的顺序。

1. 时间戳

时间戳能以各种形状和大小出现，这取决于需要的精度和正确度。时间戳的目标是保证消息只在一个给定的时间段内是有效的。例如，一个系统可能只允许一个包只能在30s内是有效的，因为使用了时间戳。时间戳听起来相当简单，但也有许多问题。

从安全性的角度来看，它们很难变得足够窄以避免在窗口的有效期内的重放攻击。例如，你把窗口的有效期设置为5min，攻击者就可以在一个5min的时间段内重放这个消息。另一方面，

如果你让窗口只有3s的有效期，你可能在窗口内传输包的时间很紧张，更糟地是，两个终端之间的时钟偏差可能会使这个系统无法使用。

最后一点会导致时间戳的使用问题。计算机并不是十分精确的时间保持者。系统时间可能会由于时间硬件的不精确，系统崩溃等原因而产生偏差。通常，大多数的操作系统都提供了一种更新系统时间的机制，一般是通过网络时间协议（Network Time Protocol，NTP）。让两台计算机，尤其是远程的，都同意当前的时间是一种困难的挑战。

2. 计数器

计数器是抵抗重放攻击更为理想的解决方案。它最基本的要求是，你需要有“下一个”值应该是什么的概念。例如，你可以在一个包中存储一个LFSR状态并期望在下一个包中的是移动过的LFSR。如果你没有看到这个移动过的LFSR的值，那么就可以推断你实际上并没有收到所要求的包。一种更为简单而有用的办法是使用一个整数作为计数器。只要计数器在相同的会话中没有被重用（也没有首先改变MAC的密钥），它们可以用来把包放入上下文中。

如果一个新MAC密钥用在会话中的话（例如，从一个密钥衍生函数中衍生出来的一个MAC密钥），计数器的值由双方协商决定而且不需要是随机的或者惟一的。每个发送的包递增计数器，计数器自身包含在消息中而且是MAC函数输入的一部分。也就是说，你要对密文和计数器都使用MAC。你可以选择对计数器加密或者不加密，但这么做常常很少有安全方面的好处。接收者在做其他事情之前先检查计数器。如果计数器的值小于或等于最新的包计数器，它很有可能是一个重放并且你应该做出相应的反应。如果是相等的，你应该继续下一步。

通常，判断计数器的大小不大于你将要发送的包的最大数目是明智的做法。例如，使用一个16个字节的计数器大多数情况下是一种浪费。对于大多数的应用程序来说，一个8字节的计数器也是过多的。对于大多数的网络应用程序，一个32位的计数器足够了，但是由于条件的不同，可能会需要一个40位或者48位的计数器。

在双向传输媒介中另一种有用的使用计数器的技巧是每个方向都定义一个计数器。这可以让传入的计数器独立于传出的计数器。在实际应用中，它允许双方能在同一时刻传输数据而不必使用一个令牌传递（token passing）机制。

6.6.5 先加密再MAC

与使用加密和MAC这两种算法相关的一个常见的问题是，用什么样的顺序来应用它们？是先加密再MAC，还是先MAC再加密？也就是说，是MAC明文还是密文？从根本上来看，它们看起来可以提供相同级别的安全性，但是还有些细微的差别。

不管选择什么顺序，加密和MAC的密钥不能轻易地相关是很重要的。最简单也是最合适的解决方案是，使用一个密钥衍生函数把一个共享的（或已知的）秘密扩展成加密和MAC密钥。

1. 先加密再MAC

在这种模式下，首先对明文加密，然后MAC密文（带有一个计数器或者时间戳）。由于加密是某种明文的随机映射（需要一个合理选择的IV），所以实质上是对一个随机的消息进行MAC。这种模式一般更喜欢用在MAC不泄露关于明文信息的基础上。同时，不需要解密也能拒绝一个无效的包。

2. 先MAC再加密

在这种模式下，首先MAC明文（带有一个计数器或者时间戳），然后加密明文。由于MAC的输入不是随机的而且大多数的MAC算法（至少CMAC和HMAC）不使用IV，所以MAC的输出将是非随机的——也就是说，如果你没有使用重放保护的话。常见的异议是MAC是基于明文的，所以你是在给攻击者提供密文和明文的MAC标记。如果明文没有改变，密文是可以改变的（通过适当地选择一个IV），但是其MAC标记仍然是相同的。

但是，也应该总是把一个计数器或者时间戳包含在MAC函数输入的一部分中。由于MAC是一个PRF，所以它会抵抗这种攻击，即使明文仍然是一样的。更好的办法是，还可以和另外一个抵抗这种攻击的方法一起使用：简单地对明文和MAC标记一起进行加密。这不能防止重放攻击（你还需要对MAC函数的输入做某些变形），但完全可以把MAC标记值对攻击者隐藏起来。

这种模式的一个缺点是，它需要受害者在他能比较MAC标记以检查是否伪造之前进行解密。奇怪地是，这个缺点有一种令人惊奇的有用的优点。它不会泄露，至少对它自身来说，时间信息给攻击者。也就是说，在第一种模式中，一旦MAC失败我们就可以终止处理。这会告诉攻击者伪造会在什么时候失败。而在这种模式中，我们总是执行同样的工作来校验MAC标记。这对于攻击者多有用取决于环境。至少，对于前一种模式来说，这是一个优势。

6.6.6 加密和认证

我们将再次使用LibTomCrypt来实现一个示例系统。这个例子是针对双向信道的，其威胁向量包括保密和认证侵害以及数据流的修改，如重排序和重放。

这个示例代码使用AES-CTR实现保密性，用HMAC-SHA256来认证以及用编号的包来进行数据流保护，通过这些办法来对抗所存在的问题。这段代码并不是最佳的，但确实提供了一个有用的基本代码，当条件改变时，可以进一步修改。尤其是这段代码不是线程安全的。也就是说，如果两个线程试图在同一时刻发送数据包时，它们会破坏状态。

encmac.c:

```
#include <tomcrypt.h>
```

我们使用LibTomCrypt来提供一些程序。如果你想试图运行这个演示程序，请把它安装好。

```
#define ENCKEYLEN     16
#define MACKEYLEN     16
```

这是加密和MAC密钥的长度。加密密钥长度必须是一个有效的AES密钥长度，因为我们选择使用AES。MAC密钥长度可以任意，但从实际应用上来说，它应该不能比加密密钥长。

```
/* Our Per Packet Sizes, CTR len and MAC len */
#define CTRLEN        4
#define MACLEN        12
#define OVERHEAD      (CTRLEN+MACLEN)
```

这3个宏定义了每个包的大小。CTRLEN定义了包计数器的大小。默认的4个字节允许在溢出和流变得不能再使用之前可以发送2^{32}个数据包。从好的一方面来看，这是一个避免长时间使用相同密钥的简单方法。

MACLEN定义了我们想要存储的MAC标记的长度。默认的12个字节（96位）足够可以使伪造很困难。因为我们局限于2^{32}个包，所以一种伪造的优势还是相当小的。

除了密文之外，这两者一起构成了每个包中的OVERHEAD字节。在默认情况下，每个包扩展了16个字节。对于一个典型的1024个字节的包大小来说，这代表了一个1.5%的开销。

```
/* errors */
#define MAC_FAILED     -3
#define PKTCTR_FAILED -4
```

MAC_FAILED表示消息的MAC标记和接收者生成的不相符。例如，如果攻击者修改了负载或者计数器（或者两者兼而有之）。PKTCTR_FAILED表示一个计数器被重放了或者顺序乱了。

```
/* our nice containers */
typedef struct {
   unsigned char PktCTR[CTRLEN],
                 enckey[ENCKEYLEN],
                 mackey[MACKEYLEN];
   symmetric_CTR skey;
} encauth_channel;
```

这个结构包含了我们对一个独立的单向信道所需要的所有细节。我们有包计数器（PktCTR）、加密密钥（enckey）和MAC密钥（mackey）。我们也预先设置了一个密钥以降低处理延迟（skey）。

```
typedef struct {
   encauth_channel channels[2];
} encauth_stream;
```

这个结构只是简单地把两个单向的流封闭为一个结构。在我们的表示中，channel[0]总是传出信道，channel[1]总是传入信道。我们稍后将看到两个对等实体是怎样使用这种约定来进行通信的。

```
void register_algorithms(void)
{
   register_cipher(&aes_desc);
   register_hash(&sha256_desc);
}
```

这个函数在LibTomCrypt中注册了AES和SHA256，这样我们就可以在插件驱动函数中使用它们。

```
int init_stream(const unsigned char *masterkey,
                unsigned  masterkeylen,
                const unsigned char *salt,
                unsigned  saltlen,
                encauth_stream *stream,
                int  node)
{
```

这个函数用一个给定的主密钥和盐渍来初始化一个双向数据流。它使用PKCS #5密钥衍生对每个方向的数据来得到一对密钥。

节点参数允许我们可以交换数据流的内容。这由其中一方所使用，使得他们的传出数据流是另一方的传入数据流（反之亦然）。

```
   unsigned char tmp[2*(ENCKEYLEN+MACKEYLEN)];
   unsigned long tmplen;
   int           err;
   encauth_channel tmpswap;
   /* derive keys */
   tmplen = sizeof(tmp);
   if ((err = pkcs_5_alg2(masterkey, masterkeylen,
                          salt, saltlen,
                          16, find_hash("sha256"),
                          tmp, &tmplen)) != CRYPT_OK) {
      return err;
   }
```

这个调用衍生出两对密钥所需要的字节。我们只使用16次的PKCS #5迭代，因为我们会假设主密钥是随机选择的。

```
   /* copy keys */
   memcpy(stream->channels[0].enckey,
          tmp, ENCKEYLEN);
   memcpy(stream->channels[0].mackey,
          tmp + ENCKEYLEN, MACKEYLEN);
   memcpy(stream->channels[1].enckey,
          tmp + ENCKEYLEN + MACKEYLEN, ENCKEYLEN);
   memcpy(stream->channels[1].mackey,
          tmp + ENCKEYLEN + MACKEYLEN + ENCKEYLEN, MACKEYLEN);
```

这个代码片段从PKCS #5输出缓冲区中提取出密钥。

```
   /* reset counters */
   memset(stream->channels[0].PktCTR, 0,
          sizeof(stream->channels[0].PktCTR));
   memset(stream->channels[1].PktCTR, 0,
          sizeof(stream->channels[1].PktCTR));
```

在每一次新的会话中，我们让包计数器从0开始。

```
   /* schedule keys+setup mode */
   /* clear an IV */
   memset(tmp, 0, 16);
   if ((err = ctr_start(find_cipher("aes"), tmp,
                        stream->channels[0].enckey, ENCKEYLEN,
                        0, CTR_COUNTER_BIG_ENDIAN,
```

```
                        &stream->channels[0].skey))!= CRYPT_OK) {
      return err;
   }

   if ((err = ctr_start(find_cipher("aes"), tmp,
                        stream->channels[1].enckey, ENCKEYLEN,
                        0, CTR_COUNTER_BIG_ENDIAN,
                        &stream->channels[1].skey))!= CRYPT_OK) {
      return err;
   }
```

此时，我们已经调度了加密密钥。这意味着在我们处理密钥时，不需要运行（相对比较慢的）AES密钥调度程序来初始化CTR上下文。

```
   /* do we swap? */
   if (node != 0) {
      tmpswap            = stream->channels[0];
      stream->channels[0] = stream->channels[1];
      stream->channels[1] = tmpswap;
      zeromem(&tmpswap, sizeof(tmpswap));
   }
```

如果我们不是节点0，那么就交换数据流的内容。这使得双方可以互相交谈。

```
   zeromem(tmp, sizeof(tmp));
```

清除栈中的密钥。注意我们使用的是LTC zeromem()函数，这不会被编译器优化成（至少很可能会）一个无操作（NOP）（对于编译器来说是有效的）。

```
   return 0;
}

int encode_frame(const unsigned char *in,
                       unsigned   inlen,
                       unsigned char *out,
                       encauth_stream *stream)
{
```

这个函数通过编号、加密和应用HMAC来对一个帧（或包）进行编码。我们在输出缓冲区中存储inlen+OVERHEAD个字节。注意in和out在内存中不能重叠。

```
   int           x, err;
   unsigned char IV[16];
   unsigned long maclen;

   /* increment counter */
   for (x = CTRLEN-1; x >= 0; x--) {
      if (++(stream->channels[0].PktCTR[x])) break;
   }
```

对包计数器以big-endian格式进行递增。这和我们在前面怎样初始化CTR会话相符合，而且稍后将会变得更加方便。

```
        /* construct an IV */
        for (x = 0; x < CTRLEN; x++) {
            IV[x] = stream->channels[0].PktCTR[x];
        }
        for (; x < 16; x++) {
            IV[x] = 0;
        }

        /* set IV */
        if ((err = ctr_setiv(IV, 16,
                             &stream->channels[0].skey)) != CRYPT_OK) {
           return err;
        }
```

包计数器仅仅是CTRLEN个字节长（默认为4），一个AES CTR模式的IV是16个字节。我们对剩下的字节用0来填充，但在这个上下文中这意味着什么？

CTR计数器是big-endian模式的。前面的CTRLEN个字节是CTR IV的最高字节。后面的字节（其中保存为0的）是最低字节。这意味着当我们加密文本时，CTR计数器只在IV的低的部分递增，以防止重叠。

例如，如果CTRLEN是15并且inlen为$257 \times 16 = 4112$，那么就可能会产生问题。第一个包的后16个字节会用IV 0...0100来加密，同时第二个包的前16个字节也会用同样的IV进行加密。

回忆第4章可以知道，CTR模式只有当IV是惟一的时候才会和它所用的分组密码同样安全（假设它是很好地密钥化了的并且合理地实现了的）。在这个例子中，它们可能不是惟一的，攻击者可以利用这种重叠。

这给实现设置了一个上限。如果CTRLEN为4，我们只可以有2^{32}个包，但是每个包都可以是2^{100}个字节长。如果CTRLEN为8，可以有2^{64}个包，每个限制在2^{68}个字节长。但是，CTRLEN越长，开销越大。更长的包计数器并不总是有帮助的。从另一方面来看，如果流量很大的话，短的包计数器的效率很低。

```
        /* Store counter */
        for (x = 0; x < CTRLEN; x++) {
            out[x] = IV[x];
        }

        /* encrypt message */
        if ((err = ctr_encrypt(in, out+CTRLEN, inlen,
                               &stream->channels[0].skey)) != CRYPT_OK) {
           return err;
        }
```

此时，我们已存储了包计数器和密文在输出缓冲区中。前CTRLEN个字节是计数器，然后

是密文。

```
    /* HMAC the ctr+ciphertext */
    maclen = MACLEN;
    if ((err = hmac_memory(find_hash("sha256"),
                           stream->channels[0].mackey, MACKEYLEN,
                           out, inlen + CTRLEN,
                           out + inlen + CTRLEN, &maclen)) != CRYPT_OK)
{
       return err;
    }
```

我们组织数据的顺序并不是偶然的。人们可能想知道我们为什么不把HMAC标记放到包计数器之后。该函数调用回答了这个问题。同时，我们也可以对计数器和密文一起HMAC。

LibTomCrypt确实提供了一个hmac_memory_multi()函数，它和hmac_memory()很相似，不同的是它使用一个va_list在一个单独的函数调用中HMAC多个内存区域（非常类似于集散列表(scattergather list)）。这个函数具有很高的调用开销，因为它使用va_list函数来获得参数。

```
    /* packet out[0...inlen+CTRLEN+MACLEN-1] now
       contains the authenticated ciphertext */
    return 0;
}
```

此时，我们已经有了可以准备传输的整个包。所有的包以inlen字节的长度输入，以inlen+OVERHEAD的字节长度作为输出。

```
int decode_frame(const unsigned char *in,
                              unsigned   inlen,
                         unsigned char *out,
                        encauth_stream *stream)
{
```

这个函数解码并且认证一个编码过的帧。注意inlen是由encode_frame()创建包的大小而不是初始明文长度。

```
    int           err;
    unsigned char IV[16], tag[MACLEN];
    unsigned long maclen;

    /* restore our original inlen */
    if (inlen < MACLEN+CTRLEN) { return -1; }
    inlen -= MACLEN+CTRLEN;
```

我们把明文的长度还原以使得函数余下的部分能够和编码兼容。第一个检查是确保输入长度是有效的。如果无效，则返回−1。

```
    /* first compute the mactag */
    maclen = MACLEN;
```

```
    if ((err = hmac_memory(find_hash("sha256"),
                           stream->channels[1].mackey, MACKEYLEN,
                           in, inlen + CTRLEN,
                           tag, &maclen)) != CRYPT_OK) {
       return err;
    }

    /* compare */
    if (memcmp(tag, in+inlen+CTRLEN, MACLEN)) {
       return MAC_FAILED;
    }
```

这时，我们已经校验了HMAC标记而且它是有效的。但是，我们还没有脱离危险。这个包有可能是一个重放的或乱序的包。

调用函数可以有一个怎样处理MAC会失败（MAC failure）的选择。如果传输媒介是像以太网一样健壮的，或者底层传输协议保证了传递，例如TCP，那么在这种情况下，MAC失败很可能就是一个篡改的标志。另一方面，如果传输媒介不健壮，例如一个无线电链接或者水印，MAC失败可能只是噪声影响信号的结果。

调用函数必须要明确怎样基于应用程序的上下文来继续下去。

```
    /* compare CTR */
    if (memcmp(in, stream->channels[1].PktCTR, CTRLEN) <= 0) {
       return PKTCTR_FAILED;
    }
```

这个memcmp()执行了一种很好的big-endia包计数器的比较操作。如果包中的包计数器不大于流结构中的包计数器，那么它会返回一个<=0的值。我们允许无序的包，但只能是前向的。例如，接收包0、3、4、7和8（以这种顺序）是有效的，而包0、3、4、1、2（以这种顺序）就是无效的。

不像MAC失败，计数器错误的发生可能是由于各种合理的原因。例如，UDP包可以采用任何顺序到达，这是有效的。虽然它们大多数情况下将是有序地到达（尤其是在传统的IPv4链接中），但无序的包也并不总是攻击的标志。另一方面，重放的包通常不是传输协议的一部分。

读者也许想增加区分重放和无序包的功能（例如，使用滑动窗口）。

```
    /* good to go, decrypt and copy the CTR */
    memset(IV, 0, 16);
    memcpy(IV, in, CTRLEN);
    memcpy(stream->channels[1].PktCTR, in, CTRLEN);

    /* set IV */
    if ((err = ctr_setiv(IV, 16,
                         &stream->channels[1].skey)) != CRYPT_OK) {
       return err;
    }
```

```

    /* encrypt message */
    if ((err = ctr_decrypt(in+CTRLEN, out, inlen,
  &stream->channels[1].skey)) != CRYPT_OK) {
        return err;
    }
    return 0;
}
```

我们的测试程序将初始两个数据流（每个方向一个），并且试图对同一个包解密3次。它可能在第一次解密的时候是有效的，但在第二次和第三次时可能会失败。在第二次尝试时，它应该以一个PKTCTR_FAILED错误而失败，因为我们重放了这个包。在第三次尝试时，我们修改了负载的一个字节并且它应该以一个MAC_FAILED错误而失败。

```
int main(void)
{
    unsigned char  masterkey[16], salt[8];
    unsigned char  inbuf[32], outbuf[32+OVERHEAD];
    encauth_stream incoming, outgoing;
    int             err;

    /* setup lib */
    register_algorithms();
```

这为我们的演示程序使用LibTomCrypt来进行设置。

```
    /* pick master key */
    rng_get_bytes(masterkey, 16, NULL);
    rng_get_bytes(salt, 8, NULL);
```

这里为了得到密钥和盐渍，我们使用了一个系统RNG。在现实的应用程序中，我们需要从一些更有用的地方得到主密钥。盐渍应该以这种方式来生成。

两种衍生一个主密钥的方法可以是通过散列一个用户的口令，或者通过使用一个公钥加密机制来共享一个随机密钥。

```
    /* setup two streams */
    if ((err = init_stream(masterkey, 16,
                           salt, 8,
                           &incoming, 0)) != CRYPT_OK) {
        printf("init_stream error: %d\n", err);
        return EXIT_FAILURE;
    }
```

这初始化了我们的传入数据流。注意我们使用值0作为节点参数。

```
        /* other side of channel would use this one */
    if ((err = init_stream(masterkey, 16,
```

```
                                      salt, 8,
                                      &outgoing, 1)) != CRYPT_OK) {
        printf("init_stream error: %d\n", err);
        return EXIT_FAILURE;
    }
```

这初始化了我们的传出数据流。注意我们使用值1作为节点参数。实际上，这和我们以什么顺序取节点值没有关系，只要是一致的，就可以很好地工作。

也要注意到，通信的每一边都只要生成一个数据流结构来编码和解码。在我们的例子中，生成两个是因为对我们生成的数据即要编码也要解码。

```
    /* make a sample message */
    memset(inbuf, 0, sizeof(inbuf));
    strcpy((char*)inbuf, "hello world");
```

传统的样本消息如下。

```
    if ((err = encode_frame(inbuf, sizeof(inbuf),
                                  outbuf, &outgoing)) != CRYPT_OK) {
       printf("encode_frame error: %d\n", err);
       return EXIT_FAILURE;
    }
```

这时，outbuf[0...sixeof(inbuf)+OVERHEAD-1]包含这个包。通过把整个缓冲区传送给其他成员，他们可以对它进行认证和解密。

```
    /* now let's try to decode it */
    memset(inbuf, 0, sizeof(inbuf));
    if ((err = decode_frame(outbuf, sizeof(outbuf),
                                  inbuf, &incoming)) != CRYPT_OK) {
       printf("decode_frame error: %d\n", err);
       return EXIT_FAILURE;
    }
    printf("Decoded data: [%s]\n", inbuf);
```

我们首先把数组inbuf清零以证明这个程序确实是解码数据的。我们使用传入的数据流结构对缓冲区进行解码。此时，我们可以在终端中看到字符串

```
Decoed data:[hello world]
```

```
    /* now let's try to decode it again (should fail) */
    memset(inbuf, 0, sizeof(inbuf));
    if ((err = decode_frame(outbuf, sizeof(outbuf),
                                  inbuf, &incoming)) != CRYPT_OK) {
       printf("decode_frame error: %d\n", err);
       if (err != PKTCTR_FAILED) {
          printf("We got the wrong error!\n");
          return EXIT_FAILURE;
       }
    }
```

这代表一个重放的包。它应该以PKTCTR_FAILED失败，而且我们应该能在终端上看到

```
decode_frame error: -4
```

```
    /* let's modify a byte and try again */
    memset(inbuf, 0, sizeof(inbuf));
    outbuf[CTRLEN] ^= 0x01;
    if ((err = decode_frame(outbuf, sizeof(outbuf),
                            inbuf, &incoming)) != CRYPT_OK) {
       printf("decode_frame error: %d\n", err);
       if (err != MAC_FAILED) {
          printf("We got the wrong error!\n");
          return EXIT_FAILURE;
       }
    }
```

这代表一个既是重放了的也是伪造了的消息。它应该在包计数器检查之前的MAC测试中失败。我们应该能在终端上看到

```
decode_frame error: -3
```

```
    return EXIT_SUCCESS;
}
```

这个演示只是表示没有完全优化的代码。根据其使用的上下文，可以有几种改进的办法。

第一种有用的优化是确保密文以一个16个字节的边界进行对齐。这使得LibTomCrypt程序能够安全地使用字对齐的异或操作来执行CTR加密。一个简单地实现它的方法是在包计数器和密文之间用0字节来填充消息（包含它作为MAC输入的一部分）。

第二种优化需要LibTomCrypt是怎样工作的知识。CTR结构很好地暴露了IV，这意味着我们能够直接设置IV而不需要使用ctr_setiv()来修改它。

第三种优化也是一种安全方面的优化。通过让代码是线程安全的，我们能够一次解码或编码多个数据包。这可以让包计数器和一个滑动窗口一起使用来确保即使线程是无序执行的，我们也有理由确定解码器会接受它们。

6.7 常见问题

下面的常见问题，由本书的作者所回答，它们即可以用来测试你对本章所出现的概念的理解，也可以帮助你在现实生活中实现这些概念。如果希望作者解答你的问题，请浏览www.syngress.com/solutions，然后点击“Ask the Author”表单。

问：什么是MAC函数？

答：MAC或者消息认证码函数就是一个接受一个秘密密钥和消息并把它化简为一个MAC标记的函数。

问：什么是MAC标记？

答：标记是一个短的位串，它用来证明秘密密钥和消息是共同通过MAC函数处理的。

问：那是什么意思？认证是什么意思？

答：能够证明消息和秘密密钥能组合在一起产生标记可以直接说明一件事：即密钥的所有

者担保或者简单地想要传达一个未经修改的原始消息。一个没有秘密密钥的伪造者应该没有相当的利益来对消息产生可验证的MAC标记。简而言之，一个MAC函数的目标就是能够断定如果MAC标记是正确的，那么消息就是完整的并且在传输过程中没有被修改。因为只有有限的成员（一般只有一个或两个）有秘密密钥，所以消息的所有权相当明显。

问：有哪些标准？

答：目前有两种可以值得考虑的针对MAC函数的NIST的标准。CMAC标准是SP 800-38B，其定义了一种把一个分组密码转化为一个MAC函数的方法。HMAC标准是FIPS-198，其定义了一种把一个散列函数转化为一个MAC的方法。一个较老的标准，FIPS-113，定义了使用DES的CBC-MAC（CMAC的前身），它应该被认为是不安全的。

问：我可以使用CMAC或者HMAC吗？

答：当密钥及实现都是安全的时候，CMAC和HMAC都是安全的。CMAC对非常短的消息更加有效。它对于已经有了一个分组密码并且空间受限的情况下，也是很理想的。HMAC对更大的消息更加有效，而且当已经有了一个散列的时候是很理想的。当然，你应该选择那个和你遵循的标准相符的MAC。

问：什么是利益（advantage）？

答：在我们的讨论中，我们已经多次看到了利益（advantage）这个术语。实质上，攻击者的利益就是，一个伪造者通过对前面已认证了的消息的分析来得到伪造的概率。例如，在CMAC的例子中，对于CMAC-AES其利益大概接近于$(mq)^2/2^{126}$——其中m是认证消息的数目，q是每个消息中AES分组的数目。当比例接近于1，那么一个成功伪造的概率也接近于1。

在这种上下文中的利益和对称加密中的有些不同。2^{-40}的利益和使用一个40位的加密密钥不同。对MAC的攻击必须是在线的。意思是说，攻击者只有一次猜对MAC标记的机会。在后者的上下文中，攻击者能够离线猜出加密密钥并且不会冒暴露的风险。

问：密钥长度在MAC函数的安全性中发挥什么样的作用？

答：密钥长度在MAC函数中的作用和它在对称加密中的作用相同。密钥越长，密钥穷举攻击所花的时间就越长。如果攻击者能猜出一个消息，那么他也能够伪造消息。

问：MAC标记的长度在MAC函数的安全性中发挥什么样的作用？

答：MAC标记的长度常常是变化的（至少在HMAC和CMAC中是这样），而且可以限制MAC函数的安全性。标记越短，伪造者就越有可能正确地地猜到它。不像散列函数，生日悖论攻击不起作用。因此，对于某些特殊的应用来说，短的MAC标记常常是非常安全的。

问：我该怎样协调密钥长度、MAC标记长度和利益这三者？

答：你的密钥长度应该越大越好。使用短的密钥通常是没有多少实际价值的。例如，用88位的0来填充一个AES-128密钥实际上把它简化到了40位的密钥，也许看起来它需要很少的资源。实际上，它并不能节省时间或者空间而且减弱了系统。理想情况下，对于一个w位长度的MAC标记，会给攻击者不超过2^{-w}的利益。例如，如果你准备用CMAC-AES发送2^{40}个消息分组，攻击的利益不小于2^{-46}。在这个例子中，一个长于46位的标记实际上是浪费，因为你已经到达了第2^{40}个消息分组。从另一方面来说，如果你发送很少数量的消息分组，其利益会非常小而且其标记长度应该调整到适合带宽的需求。

问：我为什么不能用hash（key ‖ message）来作为一个MAC函数？

答：这种结构并不抗离线攻击而且也容易受消息延长攻击。在这种机制下伪造消息是很容易的。

问：什么是重放攻击？

答：当你把一个大的消息分成一些更小的独立块（例如包）时，重放攻击就会发生。攻击者可以利用这样一个事实，除非你把包的序号关联起来，否则攻击者就可以简单地通过重新排序包的序号来改变消息的含义。虽然每个单独的包都已被认证，而且也没有被修改。这样，这种攻击就不容易被发现。

问：为什么要关心重放攻击？

答：如果没有重放保护，攻击者可以改变全部消息的含义。通常，这意味着攻击者能够重新发送语句或者命令。例如，一个攻击者可以重新发送由一个远程登录shell所发送的shell命令。

问：怎样抵抗重放攻击？

答：最明显的解决方案是找到一种方法，可以把组成消息的包序号互相关联起来。最明显的解决方案是时间戳计数器和简单的递增计数器。在这两种情况中，计数器是认证消息的一部分。基于前面已经认证的计数器的过滤机制，可以阻止攻击者重新发送一个旧的包或者无序地发送它们。

问：怎样处理包丢失或者重排序？

答：有时候，包丢失和重排序是通信传输媒介的一部分。例如，UDP是一个容忍包丢失的不严格的协议。即使包没有丢失，它们也不会保证以任何特定的顺序到达（这常是一个警告，在大多数的网络中都不会发生）。无序的UDP在非拥塞的IPv4网络中相当少。错误的意思取决于应用的环境。如果你正在使用UDP（或者其他不严格的传输媒介），包丢失和重排序通常并不是恶意的行为。拒绝包的最好实践可能是发送一个同步消息，并且继续这个协议。注意攻击者可能利用再同步步骤让受害者生成认证的消息。在一个相对稳定的媒介中，例如TCP，包丢失和重排序通常是一种恶意干涉的标志并且应该当成敌人来看。这时最通常的做法是抛弃这个连接（一般，人们争论说这是一种拒绝服务攻击（DoS）向量。但是，任何具有在你的另一台主机之间修改包能力的人也能够简单地过滤所有的数据包）。通过采取对它的警惕，就不会增加威胁。在这两种情况中，不管把错误看作是恶意的还是良性的，包都应该被丢弃并且不能进一步地在协议栈上解释。

问：哪些算法库提供MAC函数？

答：LibTomCrypt给C开发者提供了一个高度模块化的HMAC函数。Crypto++给C++开发者提供了相似的功能。有限的HMAC支持也可以在OpenSSL中找到。LibTomCrypt也提供CMAC的模块化的支持。在写作本书的时候，Crypto++和OpenSSL都不支持CMAC。我们说“模块化”意思是指HMAC和CMAC的实现并不是和某种特定的算法绑定的。例如，LibTomCrypt中的HMAC代码可以使用任何LibTomCrypt提供的散列函数而不改变API。这使得未来的更新可以用一种更省时，流水线的方式来实现。

问：MAC函数的专利保护是怎样的？

答：HMAC和CMAC都是专利免费的而且可以用于任何目的。各种其他的MAC函数，例如PMAC，虽然受专利保护但它也不是标准。

第 7 章

加密和认证模式

本章解决方案：

- 加密和认证模式
- 安全目标
- 标准
- GCM和CCM模式的设计
- 总结

☑ 总结

☑ 快速查找解决方案

☑ 常见问题

7.1 简介

在第6章中，我们已经看到如何使用消息认证码（MAC）函数来保证双方或多方之间消息的认证。MAC函数采用一个消息和秘密密钥作为输入，并且产生一个MAC标记作为输出。这个标记和消息一起，可以被任何拥有相同秘密密钥的成员校验。

我们也看到了MAC函数是怎样集成于各种应用程序中以避免各种攻击的。即如果一个攻击者能伪造消息，他必须执行我们可以而他不可以执行的任务。我们也看了如何使一个为了方便而分成一些小包的消息安全。最后，我们的示例程序把加密和认证组合在一个帧编码器中以提供保密性和认证。尤其是我们使用了PKCS #5，一个密钥衍生函数来接受一个主秘密密钥，并产生一个用于加密的密钥和另外一个用于MAC函数的密钥。

如果我们有某个函数F（K, P），它接受一个秘密密钥K和消息P，并返回分别对应于密文和MAC标记的二元组（C, T），那么这会不会不太好？我们不需要创建或者另外提供两个秘密密钥来实现加密和认证这两个目标，而是让这个过程遵循某个封装的标准。

7.1.1 加密和认证模式

本章介绍一个相对比较新的标准集，在密码学的世界中它们被称为加密和认证模式（encrypt and authenticate modes）。这些操作模式把加密和认证的任务封装到一个单独的处理过程中。这些模式的用户只要简单地传入一个单独的密钥、IV（或者nonce）和明文，然后这些模式会产生密文和MAC标记。由于把这两个任务组合到一个单独的步骤里，所以整个操作是非常容易实现的。

产生这些模式的催化剂主要有两个。第一个是为了释放把这些模式组合在一起时所能产生的任何性能上的好处。第二个是为了让认证更加吸引那些想要忽略它的开发者们。你很有可能

会找到一个可以加密数据的产品，却很少能找到一个能认证数据的产品。

7.1.2 安全目标

加密和认证模式的安全目标是保证消息的保密性和真实性。理想情况下，攻破了其中的一个也不能削弱另一个。为了达到这些目标，大多数组合模式都要求一个足够长的秘密密钥使得攻击者无法猜到它。每次调用时，它们也需要一个惟一的IV以确保重放攻击是不可能的。在这个上下文（context）中，这些惟一的IV通常叫做*nonce*。术语*nonce*实际上来自N_{once}，它的意思是使用N一次且仅一次。

在本章的后面我们可以看到，当秘密密钥是随机生成的时候，可以把*nonce*作为一个包计数器来使用。这使得它们可以很容易地集成到现有的协议中。

7.1.3 标准

即使加密和认证模式是相对比较新的，但仍然有一些不错的描述它们的设计标准。在2004年5月，NIST指定CCM为SP 800-38C标准，这是第一个NIST的加密和认证模式。它被指定为一种用于分组密码的操作模式，打算和NIST的分组密码，例如AES一起使用的。CCM是作为一种设计竞赛的结果而被选中的，从中挑出了许多提议。当时很有可能获胜的竞争者有伽罗瓦计数器模式（GCM，Galois Counter Mode）、EAX模式和CCM。

GCM最初设计是用于各种无线标准中的，例如802.16（WiMAX），然后才提交到NIST以参赛的。GCM虽然还不是一个NIST标准（它被提议作为SP 800-38D），但由于它用于IEEE无线标准中，所以它是一个需要了解的很好的算法。GCM通过大规模的可并行化来努力达到硬件上的性能要求。在软件中，我们稍后将看到，通过合理地使用处理器的缓存，GCM可以达到很高的性能水平。

最终，EAX在遵循了CCM模式之后被提出以解决CCM设计中的一些缺点。更为特别的是，EAX模式在它所使用的方式以及努力争取更高的性能（这被证明在实际应用中不是真的）方面上具有更好的灵活性。EAX模式实际上是一个精心构造的围绕CTR加密模式和CMAC认证模式的模式。这使得对它的安全分析很容易，而且其设计更值得关注。不幸地是，EAX还没有，至少目前还没有，被考虑定为标准。尽管如此，EAX仍然是一个值得知道和理解的模式。

7.2 设计与实现

我们将考虑这3个流行算法的设计、实现以及优化。我们将首先研究GCM算法，它已实际应用于IEEE 802标准系列中。读者应该对这个设计具有特别的兴趣，因为它也有可能成为NIST标准。GCM之后，我们将研究CCM的设计，它是在写作本书时仅有的NIST标准模式。CCM即高效又安全，这使得它是一个值得使用和了解的模式。

7.2.1 额外的认证数据

这3个算法都包含了一个叫做额外的认证数据（Additional Authentication Data，AAD，在CCM中也称为报头数据（header data））的输入。这使得实现者可以包含和密文一起的数据，

而且必须被认证但不需要加密。例如，元数据（metadata），如包计数器、时间戳、用户和主机名，等等。

这3种模式的AAD都是惟一的，而且处理也各不相同。特别是EAX具有最灵活的AAD处理，而GCM和CCM更加严格。3种模式都接受空的AAD串，这允许开发者在不需要AAD工具的时候可以忽略它们。

7.2.2 GCM的设计

GCM（Galois Counter Mode）是由David McGraw和John Viega所设计的。它是用于安全目的的通用散列（universal hashing）和CTR模式加密的产品。设计GCM的最初动机是为了快速的硬件实现。正因为如此，GCM才使用了$GF(2^{128})$上的乘法，这在典型的FPGA和其他硬件中可以高效地实现。

为了更好地讨论GCM，我们不得不阐明实现者最糟糕的恶梦——位排序（bit ordering）。即哪一位最高位，它们怎样排序，等等。可以看出GCM在这方面并不是最简单的设计。一旦我们了解了伽罗瓦域数学（Galois field math），GCM剩下的部分相对来说就比较容易说明了。

1. GCM GF(2)数学

GCM使用了域$GF(2^{128})[x]/v(x)$上的乘法来执行一个叫做GHASH的函数。实际上，GHASH是通用散列的一种形式，我们将在下面讨论它。这里执行的乘法和AES分组密码中使用的乘法没有任何本质上的区别。它们仅有的区别是域的大小和所用的不可约多项式。

GCM使用了一种初看上去似乎和平时不一样的位排序。它不是把多项式的系数从最低位开始向高位进行存储，而是反过来存储。例如，从AES中我们可以看到多项式$p(x)=x^7+x^3+x+1$可以用0x8B来表示。而在GCM符号表示中，这些位被逆过来了。在GCM符号表示中，x^7是0x01而不是0x80，因此多项式$p(x)$将用0xD1来表示。实际上，这些字节是little-endian形式。这些字节自身以big-endian格式排列，这更为复杂了。也就是说，字节数15是最低字节，字节数0是最高字节。

乘法程序可以用如下的函数来实现。

```
static void gcm_rightshift(unsigned char *a)
{
   int x;
   for (x = 15; x > 0; x--) {
       a[x] = (a[x]>>1) | ((a[x-1]<<7)&0x80);
   }
   a[0] >>= 1;
}
```

在GCM中，它叫做右移运算（right shift operation）。从数值上来看，它等于一个左移（乘以2），但由于我们以相反的方向排列每个字节中的位，所以我们使用一个右移来执行它。我们从字节15向下移动到字节0。

```
static const unsigned char mask[] = {
    0x80, 0x40, 0x20, 0x10, 0x08, 0x04, 0x02, 0x01
};
static const unsigned char poly[] = { 0x00, 0xE1 };
```

mask是一种简单地逆序屏蔽字节中位的方法。数组poly是多项式的最低字节，第一个元素为0，第二个元素是多项式的字节。在本例中，0xE1映射为$p(x)=x^{128}+x^7+x^3+x+1$，其中x^{128}这一项是隐含的。

```
void gcm_gf_mult(const unsigned char *a,
                 const unsigned char *b,
                       unsigned char *c)
{
   unsigned char Z[16], V[16];
   unsigned x, y, z;

   memset(Z, 0, 16);
   memcpy(V, a, 16);
   for (x = 0; x < 128; x++) {
       if (b[x>>3] & mask[x&7]) {
          for (y = 0; y < 16; y++) {
              Z[y] ^= V[y];
          }
       }
       z      = V[15] & 0x01;
       gcm_rightshift(V);
       V[0] ^= poly[z];
   }
   memcpy(c, Z, 16);
}
```

这个程序实现了GCM所选择的Galois域上的$c=ab$运算。它实际上和用于AES中的乘法是同一个算法，但这里我们使用一个字节数组来代表多项式。我们使用Z来存储积。我们使用V作为a的一个拷贝，根据b的位我们可以对它进行加倍以及选择性的把它和Z进行相加。

这个乘法函数在数值上实现了我们所需要的，但它非常慢。幸运地是，存在不止一种乘以域元素的方法。我们将在实现阶段中看到，使用一个基于表的乘法程序会得到更多的好处。

对于想要了解更多内容的读者也许想看看LibTomCrypt中的GCM源代码来了解根据配置不同而选择性地使用的各种技巧。除了前一个程序之外，LibTomCrypt提供了一种对gcm_gf_mult()的变形程序（参见LibTomCrypt中的src/encauth/gcm/gcm_gf_mult.c），它使用了一个对所有字窗口化了的乘法（Darrel Hankerson，Alfred Menezes，Scott Vanstone，"Guide to Elliptic Curve Cryptography（中文译为《椭圆曲线密码学导论》)"，p.50，Algorithm 2.36）。这在GCM的设置阶段很重要，即使当我们使用一个基于表的乘法程序来处理大量数据时。在把基于表的乘法程序展现给你看之前，我们必须先告诉你使得使用这种方法成为可能的对GCM的限制。

2. 通用散列

通用散列是一种创建一个函数$f(x)$使得对于不同的值x和y，$f(x)=f(y)$的概率为任意一种合理的随机函数中的概率的方法。这种通用散列的一个最简单的例子是映射

$$f(x) = (ax + b \bmod p) \bmod n$$

其中a和b都是随机值，p和n是随机的素数（$n<p$）。通用MAC函数，例如GCM中的（以及其他算法，例如Daniel Bernstein's Poly1305）使用这种映射的一种变形来实现一个安全的MAC函数

$$H[i] = (H[i-1]*K) + M[i]$$

其中，最后一个H[i]的值就是标记，*K*是一个有限域中的一个单元而且也是秘密密钥，M[i]是消息的一个分组。乘法和加法必须在一个相当大的有限域上执行（例如，2^{128}个单元或更多）。在GCM中，我们将创建MAC功能，它叫做GHASH，这种机制使用我们的GF(2^{128})乘法程序。

3. GCM的定义

整个GCM算法可以由一系列的等式来指定。首先，我们来定义用于等式中的各种符号（如图7-1所示）。

- 令*K*表示秘密密钥；
- 令*A*表示额外的认证数据，*A*中有*m*个分组的数据；
- 令*P*表示明文，*P*中有*n*个分组的数据；
- 令*C*表示密文；
- 令*Y*表示CTR计数器；
- 令*T*表示MAC标记；
- 令*E*（*K*,*P*）表示用秘密密钥*K*和分组密码*E*（例如*E*=AES）对*P*进行加密；
- 令IV表示用于待处理消息的*IV*。

输入：

P: 明文
K: 秘密密钥
A: 额外的认证数据
IV: GCM初始向量

输出：

C: 密文
T: MAC标记

1. $H = E(K, 0)$
2. If length(IV) = 96
   1. $Y_0 = IV \,||\, 0^{31}1$

   else

   2. Y_0 = GHASH(H, {}, IV)
3. $Y_i = Y_{i-1} + 1$, for i = 1, ..., n
4. $C_i = P_i$ XOR E(K, Y_i), for i = 1, ..., n − 1
5. $C_n = P_n$ XOR E(K, Y_n), truncated to the length of P_n
6. T = GHASH(H, A, C) XOR E(K, Y_0)
7. 返回 C 和 T

图7-1 GCM数据处理

第一步是生成通用MAC的密钥*H*，它主要用于GHASH函数。接着，我们需要一个用于CTR模式的IV。如果用户提供的IV是96位长，我们就直接使用它，并在它的后面填充31个0和1个1。否则，我们就对IV应用GHASH函数并把返回值作为CTR的IV来使用。

一旦有了*H*和初始值Y_0，我们就能够对明文加密。加密以CTR模式执行，并以big-endian形

式使用计数器。奇怪地是，计数器每个字节的位是以正常的顺序来使用的。最后一个密文分组，如果它不能填满一个分组的话，它将不会被扩展。例如，如果P_n是32位，$E(K, Y_n)$的输出会被截短为32位，并且C_n是这两个值的32位的异或。

最后，通过对额外的认证数据和密文执行GHASH函数来产生MAC标记。GHASH的输出然后与对初始Y_0值的加密进行异或。下面，我们来看GHASH函数（如图7-2所示）。

```
输入：
    H: 秘密参数（衍生自秘密密钥）
    A: 额外的认证数据（m个分组）
    C: 密文（也用做一个额外的输入源，n个分组）
输出：
    T: GHASH输出
  1. X_0 = 0
  2. For i from 1 to m do
       1. X_i = (X_{i-1} XOR A_i) * H
  3. For i from 1 to n do
       1. X_{i+m} = (X_{i+m-1} XOR C_i) * H
  4. T = (X_{m+n} XOR (length(A)||length(C)) * H
  5. 返回  T
```

图7-2 GCM GHASH函数

GHASH函数压缩额外的认证数据和密文为最终的MAC标记。乘上H的乘法是一个在前面提到过的$GF(2^{128})[x]$上的乘法。其长度编码是一些big-endian格式的64位的互相连接的串。A的长度保存在前8个字节中，C的长度保存在后8个字节中。

7.2.3 GCM的实现

虽然GCM的描述是不错的而且简洁的，但其实现却不是。首先，乘法运算需要仔细地优化才能得到相当好的性能。其次，IV、AAD和明文处理的灵活性需要仔细的状态转换。一开始，我们打算写一个从本文中提取的实现，但是，我们后来决定如果简单地使用一个现有的优化过了的实现，读者可能会更好地理解。

为了演示GCM，我们使用LibTomCrypt中的实现。这个实现是公开的，可以免费在工程的主页上访问，而且是优化过了的，很容易理解。我们将省略一部分管理代码以减少代码列表的大小。强烈建议读者使用LibTomCrypt（或者类似的算法库）中的程序，而不要用自己实现的。

1. 接口

我们的GCM接口有几个将按照顺序进行讨论的函数。高级别的抽象使得我们能够使用GCM的实现以达到GCM规范中说明的完全灵活性。我们将要讨论的函数是：

(1) gcm_gf_mult()　　通用$GF(2^{128})[x]$乘法

(2) gcm_mult_h()　　乘以H（通常是优化了的，因为在设置之后H是固定的）

(3) gcm_init()　　初始一个GCM状态
(4) gcm_add_iv()　　把IV数据加到GCM状态中
(5) gcm_add_aad()　　把AAD加到GCM状态中
(6) gcm_process()　　把明文加到GCM状态中
(7) gcm_done()　　结束GCM状态并返回MAC标记

这些函数组合起用以允许调用函数能通过GCM算法来处理数据。对任何消息，函数3～7是以这种顺序去调用以处理数据。即必须在AAD之前加上IV，在明文之前加上AAD。GCM不允许以其他的顺序处理不同的数据元素。例如，你不能在IV之前就加上AAD。这些函数可以被多次调用，只要出现的顺序是正确的。例如，你可以在第一次调用gcm_add_aad（）之前调用gcm_add_iv()两次。

所有的函数都使用*gcm_state*结构，它包含GCM算法当前的工作状态。它完全决定函数应该做什么，这使得函数是完全线程安全的（如图7-3所示）。

```
typedef struct {
   symmetric_key        K;
   unsigned char        H[16],    /* multiplier */
                        X[16],    /* accumulator */
                        Y[16],    /* counter */
                        Y_0[16],  /* initial counter */
                        buf[16];  /* buffer for stuff */

   int                  cipher,   /* which cipher */
                        ivmode,   /* Which mode is the IV in? */
                        mode,     /* mode the GCM code is in */
                        buflen;  /* length of data in buf */

   ulong64              totlen,   /* 64-bit counter used for IV and AAD */
                        pttotlen; /* 64-bit counter for the PT */

#ifdef GCM_TABLES
   unsigned char        PC[16][256][16];  /* 16 tables of 8x128 */
#endif
} gcm_state;
```

图7-3　GCM状态结构

我们可以看到，这个状态包含许多成员。表7-1解释了它们的功能。

表7-1　gcm_state成员及它们的功能

成员名称	目　的
K	调度的密码密钥，用于对计数器加密
H	GHASH乘数值
X	GHASH累加器
Y	CTR模式计数器值（文本处理时递增）
Y_0	用于对GHASH输出进行加密的初始计数器值
buf	用在好几个地方，例如，存储加密了的计数器值

（续）

成员名称	目　　的
cipher	GCM所用的分组密码的ID
ivmode	指定我们是否使用一个短的IV。如果IV大于12个字节，它是非零值
mode	GCM所处的当前模式。可以为以下值之一： GCM_MODE_IV GCM_MODE_AAD GCM_MODE_TEXT
buflen	数组buf中当前的数据长度
totlen	IV和AAD数据的总长度
pttotlen	明文的总长度
PC	一个16×256×16的表，$PC[i][j][k]$是在$GF(2^{128})[x]$中的$H*j*x^{8i}$的第k个字节。这个表是基于秘密的H值由gcm_init()预计算的，它用来加速GHASH函数所要求的乘H的运算

PC表是可选的，仅当GCM_TABLES在构建时定义了才会包含。我们稍后将看到，它能快速地加速通过GHASH处理数据的过程；但是，它需要一个64KB的表，这在各种嵌入式平台上通常是不允许的。

2. GCM通用乘法

下面的代码实现了GCM所需的通用$GF(2^{128})[x]$乘法。它设计可用于任意乘数，而且并没有针对GHASH中乘以一个单独的值（H）的用法类型进行优化。

gcm_gf_mult.c:

```
/* this is x*2^128 mod p(x) ... the results are 16 bytes
 * each stored in a packed format.  Since only the
 * lower 16 bits are not zero'ed I removed the upper 14 bytes */
const unsigned char gcm_shift_table[256*2] = {
0x00, 0x00, 0x01, 0xc2, 0x03, 0x84, 0x02, 0x46,
0x07, 0x08, 0x06, 0xca, 0x04, 0x8c, 0x05, 0x4e,
0x0e, 0x10, 0x0f, 0xd2, 0x0d, 0x94, 0x0c, 0x56,
0x09, 0x18, 0x08, 0xda, 0x0a, 0x9c, 0x0b, 0x5e,
<snip>
0xb5, 0xe0, 0xb4, 0x22, 0xb6, 0x64, 0xb7, 0xa6,
0xb2, 0xe8, 0xb3, 0x2a, 0xb1, 0x6c, 0xb0, 0xae,
0xbb, 0xf0, 0xba, 0x32, 0xb8, 0x74, 0xb9, 0xb6,
0xbc, 0xf8, 0xbd, 0x3a, 0xbf, 0x7c, 0xbe, 0xbe };
```

这个表包含了对于k的所有的256个值，$k*x^{128}$ mod $p(x)$的余数。由于$p(x)$的值是稀疏的，所以只有余数的低2个字节是非零的。正因为如此，我们可以压缩这个表。每对字节是给定的k值的余数的低2个字节。例如，gcm_shift_table[3]和gcm_shift_table[4]是$2*x^{128}$ mod $p(x)$值的最低字节。

这个表只有当LTC_FAST被定义时才使用。这个定义告诉实现使用一个快速的并行在字一级的异或操作，而不是在字节一级。在我们的例子中，我们可以利用它来让这个通用乘法执行得更快。

```
#ifndef LTC_FAST
/* right shift */
static void gcm_rightshift(unsigned char *a)
{
```

```
    int x;
    for (x = 15; x > 0; x--) {
        a[x] = (a[x]>>1) | ((a[x-1]<<7)&0x80);
    }
    a[0] >>= 1;
}
```

这个函数使用GCM约定执行右移运算（乘以x）。

```
/* c = b*a */
static const unsigned char mask[] =
   { 0x80, 0x40, 0x20, 0x10, 0x08, 0x04, 0x02, 0x01 };
static const unsigned char poly[] =
   { 0x00, 0xE1 };


/**
  GCM GF multiplier (internal use only)  bitserial
  @param a   First value
  @param b   Second value
  @param c   Destination for a * b
 */
void gcm_gf_mult(const unsigned char *a,
                 const unsigned char *b,
                       unsigned char *c)
{
   unsigned char Z[16], V[16];
   unsigned x, y, z;

   zeromem(Z, 16);
   XMEMCPY(V, a, 16);
   for (x = 0; x < 128; x++) {
       if (b[x>>3] & mask[x&7]) {
          for (y = 0; y < 16; y++) {
              Z[y] ^= V[y];
          }
       }
       z     = V[15] & 0x01;
       gcm_rightshift(V);
       V[0] ^= poly[z];
   }
   XMEMCPY(c, Z, 16);
}
```

这是慢的位序列方法。出于可移植问题的考虑，我们使用LibTomCrypt函数zeromem（类似于memset）和XMEMCPY（默认是memcpy）。许多小型C平台并没有完整的C库函数。这些函数（以及宏）可以使得开发者能以一种安全的方式解决这种限制。

```
#else

/* map normal numbers to "ieee" way ... e.g. bit reversed */
#define M(x) (((x&8)>>3) | ((x&4)>>1) | ((x&2)<<1) | ((x&1)<<3))
#define BPD (sizeof(LTC_FAST_TYPE) * 8)
#define WPV (1 + (16 / sizeof(LTC_FAST_TYPE)))
```

这是一些我们在快速通用乘法函数中使用的宏。M()宏把一个4位字节映射成GCM的约定（即逆序）。

LTC_FAST_TYPE符号是在LibTomCrypt中定义的一种数据类型，它表示一个最佳的数据类型，即在长度上是8位的倍数。例如，在32位的平台上，它是一个*unsigned long*。这个数据类型可以完美地和*unsigned char*数据类型相转换。它用来允许并行异或操作。

BPD宏是每个LTC_FAST_TYPE的字节数。显然，它只有在CHAR_BIT为8的时候才有效，这也是为什么LTC_FAST默认没有开启的原因。WPV宏是每128位值的字数再加上一个字。

```
/**
  GCM GF multiplier (internal use only)  word oriented
  @param a   First value
  @param b   Second value
  @param c   Destination for a * b
 */
void gcm_gf_mult(const unsigned char *a,
                 const unsigned char *b,
                       unsigned char *c)
{
   int i, j, k, u;
   LTC_FAST_TYPE   B[16][WPV],
                  tmp[32 / sizeof(LTC_FAST_TYPE)],
                   pB[16 / sizeof(LTC_FAST_TYPE)],
                   zz, z;
   unsigned char pTmp[32];
```

数组*B*包含对于*k*=0...15，计算的值*ka*。它允许我们使用一个查表操作来执行4×128的乘法运算。数组*tmp*包含其乘积（在它被化简之前）。数组*pB*包含以GCM中的位序进行加载转换的*b*的拷贝。

```
   /* create simple tables */
   zeromem(B[0],       sizeof(B[0]));
   zeromem(B[M(1)],    sizeof(B[M(1)]));

#ifdef ENDIAN_32BITWORD
   for (i = 0; i < 4; i++) {
       LOAD32H(B[M(1)][i], a + (i<<2));
       LOAD32L(pB[i],      b + (i<<2));
   }
#else
   for (i = 0; i < 2; i++) {
       LOAD64H(B[M(1)][i], a + (i<<3));
       LOAD64L(pB[i],      b + (i<<3));
   }
#endif
```

上面的代码把*a*和*b*的字节分别加载到各自的数组中。想要了解更多内容的读者也许会注意到，我们以big-endian宏加载*a*，用little-endian宏加载*b*。*a*的值之所以用big-endian形式加载，是为了遵循GCM规范。*b*的值是以相反的形式加载，这样我们就能够使用一种更加简单的数字提取表示。

实际上，我们可以把两者都以big-endian形式来加载，并且几乎重写了我们取4位字节来补充的顺序。

```
   /* now create 2, 4 and 8 */
   B[M(2)][0] = B[M(1)][0] >> 1;
```

```
B[M(4)][0] = B[M(1)][0] >> 2;
B[M(8)][0] = B[M(1)][0] >> 3;
for (i = 1; i < (int)WPV; i++) {
B[M(2)][i] =(B[M(1)][i-1] << (BPD-1)) | (B[M(1)][i] >> 1);
B[M(4)][i] =(B[M(1)][i-1] << (BPD-2)) | (B[M(1)][i] >> 2);
B[M(8)][i] =(B[M(1)][i-1] << (BPD-3)) | (B[M(1)][i] >> 3);
}
```

这段代码创建了相应于ax、ax^2和ax^3的条目。注意我们不执行任何约简。这是为什么WPV有一个附加字的原因，我们处理的是多于128位的值。

```
/*  now all values with two bits which are
 *  3, 5, 6, 9, 10, 12 */
for (i = 0; i < (int)WPV; i++) {
   B[M(3)][i]  = B[M(1)][i] ^ B[M(2)][i];
   B[M(5)][i]  = B[M(1)][i] ^ B[M(4)][i];
   B[M(6)][i]  = B[M(2)][i] ^ B[M(4)][i];
   B[M(9)][i]  = B[M(1)][i] ^ B[M(8)][i];
   B[M(10)][i] = B[M(2)][i] ^ B[M(8)][i];
   B[M(12)][i] = B[M(8)][i] ^ B[M(4)][i];

/* now all 3 bit values and the only 4 bit value:
 * 7, 11, 13, 14, 15 */
   B[M(7)][i]  = B[M(3)][i] ^ B[M(4)][i];
   B[M(11)][i] = B[M(3)][i] ^ B[M(8)][i];
   B[M(13)][i] = B[M(1)][i] ^ B[M(12)][i];
   B[M(14)][i] = B[M(6)][i] ^ B[M(8)][i];
   B[M(15)][i] = B[M(7)][i] ^ B[M(8)][i];
}
```

这两段代码构造余下的条目。我们首先构造只有2位集的值（3、5、6、9、10和12），然后构造含有3位集的值。注意M()宏的使用，估计在编译时是一个常量。

```
zeromem(tmp, sizeof(tmp));

/* compute product four bits of each word at a time */
/* for each nibble */
for (i = (BPD/4)-1; i >= 0; i--) {
    /* for each word */
    for (j = 0; j < (int)(WPV-1); j++) {
     /* grab the 4 bits recall the nibbles are
        backwards so it's a shift by (i^1)*4 */
        u = (pB[j] >> ((i^1)<<2)) & 15;
```

这里我们提取b的四位和a相乘。注意用（i^1）来提取逆序的半位字节，因为GCM以逆序存储每个字节中的位。

```
     /* add offset by the word count the table
        looked up value to the result */
        for (k = 0; k < (int)WPV; k++) {
            tmp[k+j] ^= B[u][k];
        }
    }
```

这个循环用a乘上b的每个字中的每4个四位字节，并把它加到*tmp*中合适的偏移中去。第j个字的第i个四位字节的积被加到tmp[j...j+WPV−1]中。

```
        /* shift result up by 4 bits */
          if (i != 0) {
          for (z=j=0; j < (int)(32 / sizeof(LTC_FAST_TYPE)); j++) {
             zz = tmp[j] << (BPD-4);
             tmp[j] = (tmp[j] >> 4) | z;
             z = zz;
          }
          }
      }
```

在我们把所有关于每个字的第i个四位字节的积都加起来之后，我们把整个积（tmp）向右移4位。

```
      /* store product */
#ifdef ENDIAN_32BITWORD
      for (i = 0; i < 8; i++) {
          STORE32H(tmp[i], pTmp + (i<<2));
      }
#else
      for (i = 0; i < 4; i++) {
          STORE64H(tmp[i], pTmp + (i<<3));
      }
#endif

      /* reduce by taking most significant byte and adding the
         appropriate two byte sequence 16 bytes down */
      for (i = 31; i >= 16; i--) {
          pTmp[i-16] ^= gcm_shift_table[((unsigned)pTmp[i]<<1)];
          pTmp[i-15] ^= gcm_shift_table[((unsigned)pTmp[i]<<1)+1];
      }
```

这个约简利用了这样一个事实，即对于任意的$j>15$，kx^j mod p(x)的值和（kx^{16}）x^{j-16}同余。由于我们已经有了一个kx^{16} mod p(x)的表，所以我们能够通过一个查表操作和一个移位操作来计算（kx^{16}）x^{j-16}。这个程序按照从高字节到低字节的顺序加上积的余数。

上面代码中的每个for循环都是一次积中删除一个字节。我们通过把查表值和pTmp[$i-16$]和pTmp[$i-15$]相加来内联的执行移位操作。

```
      for (i = 0; i < 16; i++) {
          c[i] = pTmp[i];
      }

  }

  #endif
```

gcm_gf_mult()的两种实现都完成了同样的目标，并且在数值上是相等的。后一种实现在32位和64位处理器上更快，但它不是100%可移植的。它需要一个在大小上是*unsigned char*数据类型的倍数的数据类型，这并不总是存在的。

既然我们有了一个通用的乘法函数，那么我们需要为GHASH实现一个优化的乘法函数。

3. GCM优化的乘法

下面的乘法程序是优化了的，主要用来执行一个和秘密的H值相乘的运算。它利用了这样一个事实，即我们可以为乘法运算预计算一些表。

gcm_mult_h.c:

```
/**
  GCM multiply by H
  @param gcm   The GCM state which holds the H value
  @param I     The value to multiply H by
 */
void gcm_mult_h(gcm_state *gcm, unsigned char *I)
{
   unsigned char T[16];
#ifdef GCM_TABLES
   int x, y;
   XMEMCPY(T, &gcm->PC[0][I[0]][0], 16);
```

如果GCM_TABLES已被定义，我们将使用查表的方法。表PC包含了16个8×128的表，一个表示输入的每个字节，另一个表示它们各自的可能值。我们必须要做的第一件事是复制其第0个条目到T（累加器）。其余的查表值将和这个值相异或。

```
   for (x = 1; x < 16; x++) {
#ifdef LTC_FAST
       for (y = 0; y < 16; y += sizeof(LTC_FAST_TYPE)) {
           *((LTC_FAST_TYPE *)(T + y)) ^=
           *((LTC_FAST_TYPE *)(&gcm->PC[x][I[x]][y]));
       }
#else
       for (y = 0; y < 16; y++) {
           T[y] ^= gcm->PC[x][I[x]][y];
       }
#endif
```

在这里，我们看到使用LTC_FAST来优化并行的异或操作。对于输入I的每个字节，我们查找其128位的值并将它和累加器相异或。因为表中的条目已经是约简过的，所以我们的累加器在大小上不会超过128位。

```
   }
#else
   gcm_gf_mult(gcm->H, I, T);
#endif
   XMEMCPY(I, T, 16);
}
```

如果我们没有使用表，那么就用更慢的gcm_gf_mult（）来完成这个操作。

既然我们已经实现了两种乘法函数，那么就可以继续看GCM算法的其余部分了，首先从初始化程序开始。

4. GCM的初始化

第一个函数是gcm_init()，它接受一个秘密密钥并初始GCM的状态。

gcm_init.c:

```
/**
  Initialize a GCM state
  @param gcm      The GCM state to initialize
  @param cipher   The index of the cipher to use
  @param key      The secret key
  @param keylen   The length of the secret key
  @return CRYPT_OK on success
 */
```

```
int gcm_init(gcm_state *gcm, int cipher,
             const unsigned char *key,  int keylen)
{
   int           err;
   unsigned char B[16];
#ifdef GCM_TABLES
   int           x, y, z, t;
#endif

   LTC_ARGCHK(gcm != NULL);
   LTC_ARGCHK(key != NULL);

#ifdef LTC_FAST
   if (16 % sizeof(LTC_FAST_TYPE)) {
      return CRYPT_INVALID_ARG;
   }
#endif
```

这是一个简单的程序健壮性检查，以确保这段代码在LTC_FAST被定义时能实际地工作。它并没有考虑到所有的出错情形，但在实际应用中已足够了。

```
   /* is cipher valid? */
   if ((err = cipher_is_valid(cipher)) != CRYPT_OK) {
      return err;
   }
   if (cipher_descriptor[cipher].block_length != 16) {
      return CRYPT_INVALID_CIPHER;
   }

   /* schedule key */
   if ((err = cipher_descriptor[cipher].setup(key, keylen,
                                               0, &gcm->K)) !=
       CRYPT_OK) {
      return err;
   }
```

这个代码调度秘密密钥以用于GCM模式中。

```
   /* H = E(0) */
   zeromem(B, 16);
   if ((err =
        cipher_descriptor[cipher].ecb_encrypt(B, gcm->H,
                                              &gcm->K)) !=
       CRYPT_OK) {
      return err;
   }
```

我们对0串进行加密以计算秘密的乘数值H。

```
   /* setup state */
   zeromem(gcm->buf, sizeof(gcm->buf));
   zeromem(gcm->X,   sizeof(gcm->X));
   gcm->cipher   = cipher;
   gcm->mode     = GCM_MODE_IV;
   gcm->ivmode   = 0;
   gcm->buflen   = 0;
   gcm->totlen   = 0;
   gcm->pttotlen = 0;
```

这段代码把GCM状态初始化为默认的空和零状态。在这之后，我们已经准备好处理IV、AAD或者明文了（前提是GCM_TABLES没有定义）。

```
#ifdef GCM_TABLES
   /* setup tables */

   /* generate the first table as it has no shifting
    * (from which we make the other tables) */
   zeromem(B, 16);
   for (y = 0; y < 256; y++) {
        B[0] = y;
        gcm_gf_mult(gcm->H, B, &gcm->PC[0][y][0]);
   }
```

如果我们在使用表，那么首先计算最低的那个表，即对于y的所有的256个值，计算$yH \bmod p(x)$。我们使用更慢的那个乘法函数，因为此时我们还没有表。

```
   /* now generate the rest of the tables
    * based the previous table */
   for (x = 1; x < 16; x++) {
      for (y = 0; y < 256; y++) {
         /* now shift it right by 8 bits */
         t = gcm->PC[x-1][y][15];
         for (z = 15; z > 0; z--) {
             gcm->PC[x][y][z] = gcm->PC[x-1][y][z-1];
         }
         gcm->PC[x][y][0] = gcm_shift_table[t<<1];
         gcm->PC[x][y][1] ^= gcm_shift_table[(t<<1)+1];
     }
   }
```

这个代码段生成其他15个8×128的表。因为第0个8×128的表和第1个8×128的表之间惟一的区别就是多乘了一个值x^8，所以我们可以用一个简单的移位并且和约简表中的值（在gcm_gf_mult()函数中看到的）相异或来完成这一操作。我们可以重复这个过程来生成第2个、第3个、第4个表，等等。

使用这种移位约简的技巧比起老老实实地使用gcm_gf_mult()来产生所有需要的16×256个乘法要快得多。

```
#endif

   return CRYPT_OK;
}
```

这时，我们可以很好地使用GCM来处理IV、AAD或者明文了。

5. GCM的IV处理

GCM的IV控制了用于加密的CTR值的初始值，而且间接地控制了MAC标记值，因为它是基于分组密码的。如果使用了相同的密钥，那么每个包都要有一个惟一的IV。把一个包计数器用做IV是安全的，只要在使用相同的密钥时，它没有被重复使用就可以。

gcm_add_iv.c:

```
/**
  Add IV data to the GCM state
  @param gcm     The GCM state
```

```
    @param IV     The initial value data to add
    @param IVlen  The length of the IV
    @return CRYPT_OK on success
   */
  int gcm_add_iv(gcm_state *gcm,
                 const unsigned char *IV,     unsigned long IVlen)
  {
     unsigned long x, y;
     int           err;

     LTC_ARGCHK(gcm != NULL);
     if (IVlen > 0) {
        LTC_ARGCHK(IV  != NULL);
     }
```

这有点奇怪，但我们确实允许空的IV，也可以有IV==NULL。

```
     /* must be in IV mode */
     if (gcm->mode != GCM_MODE_IV) {
        return CRYPT_INVALID_ARG;
     }
```

我们必须在IV模式下调用这个函数。如果不是，那么就意味着我们调用的是AAD或者处理函数（处理明文的）。

```
     if (gcm->buflen >= 16 || gcm->buflen < 0) {
        return CRYPT_INVALID_ARG;
     }

     if ((err = cipher_is_valid(gcm->cipher)) != CRYPT_OK) {
        return err;
     }


     /* trip the ivmode flag */
     if (IVlen + gcm->buflen > 12) {
        gcm->ivmode |= 1;
     }
```

如果我们有超过12个字节的IV，那么就设置*ivmode*标志。对IV的处理是根据这个标志是否有设置而不同。

```
     x = 0;
  #ifdef LTC_FAST
     if (gcm->buflen == 0) {
        for (x = 0; x < (IVlen & ~15); x += 16) {
            for (y = 0; y < 16; y += sizeof(LTC_FAST_TYPE)) {
                *((LTC_FAST_TYPE*)(&gcm->X[y])) ^=
                *((LTC_FAST_TYPE*)(&IV[x + y]));
            }
            gcm_mult_h(gcm, gcm->X);
            gcm->totlen += 128;
        }
        IV += x;
     }
  #endif
```

如果我们可以使用LTC_FAST，那么就一次一个字的把IV中的字节加到状态中。只有当IV

有16个或者更多个IV可以加的话，我们才使用这种优化。通常，IV是很短的，因此这不太可能会被调用，但它确实允许用户在需要的时候让IV基于一个更大的串。

```
    /* start adding IV data to the state */
    for (; x < IVlen; x++) {
        gcm->buf[gcm->buflen++] = *IV++;

        if (gcm->buflen == 16) {
          /* GF mult it */
          for (y = 0; y < 16; y++) {
              gcm->X[y] ^= gcm->buf[y];
          }
          gcm_mult_h(gcm, gcm->X);
          gcm->buflen = 0;
          gcm->totlen += 128;
       }
    }
```

这段代码处理了把IV中所有剩余的字节加到状态中。它一次加一个字节，一旦我们累计到16，就把它们和状态进行异或并调用gcm_mult_h()函数。

```
    return CRYPT_OK;
}
```

这个函数处理了把IV数据加到状态中。仔细观察可以发现，它并没结束GHASH或者计算初始计数器值。那是在下一个函数中计算的，即gcm_add_aad()。

6. GCM的AAD处理

额外的认证数据（AAD）是元数据，你可以把它加到正在认证但没有加密的数据流中。它必须在IV之后以及处理明文（或密文）之前添加。如果没有数据要加到数据流中，那么AAD这一步可以跳过。一般来说，AAD是非隐私的信息，是和数据流或者包相关的数据，例如惟一的标识符或者占位标志。

gcm_add_aad.c:

```
/**
  Add AAD to the GCM state
  @param gcm       The GCM state
  @param adata     The AAD to add to the GCM state
  @param adatalen  The length of the AAD data.
  @return CRYPT_OK on success
 */
int gcm_add_aad(          gcm_state *gcm,
                const unsigned char *adata,
                      unsigned long adatalen)
{
   unsigned long x;
   int           err;
#ifdef LTC_FAST
   unsigned long y;
#endif

   LTC_ARGCHK(gcm     != NULL);
   if (adatalen > 0) {
      LTC_ARGCHK(adata  != NULL);
   }
```

```

   if (gcm->buflen > 16 || gcm->buflen < 0) {
      return CRYPT_INVALID_ARG;
   }

   if ((err = cipher_is_valid(gcm->cipher)) != CRYPT_OK) {
      return err;
   }
```

到目前为止，这还是同样的错误检查风格。我们不把它放到其他函数中的惟一原因是，确保LTC_ARGCHK宏能报告正确的函数名（它们和assert宏很相似）。

```
   /* in IV mode? */
   if (gcm->mode == GCM_MODE_IV) {
      /* let's process the IV */
```

这段代码完成对IV的处理，如果需要的话。

```
      if (gcm->ivmode || gcm->buflen != 12) {
```

如果我们已经开启了IV标记或者IV累积的长度不是12，我们就不得不对IV数据应用GHASH以产生起始*Y*值。

```
         for (x = 0; x < (unsigned long)gcm->buflen; x++) {
             gcm->X[x] ^= gcm->buf[x];
         }
         if (gcm->buflen) {
            gcm->totlen += gcm->buflen * CONST64(8);
            gcm_mult_h(gcm, gcm->X);
         }
```

此时，我们已清空了IV缓冲区并且用秘密*H*值乘上了GCM状态。

```
         /* mix in the length */
         zeromem(gcm->buf, 8);
         STORE64H(gcm->totlen, gcm->buf+8);
         for (x = 0; x < 16; x++) {
             gcm->X[x] ^= gcm->buf[x];
         }
         gcm_mult_h(gcm, gcm->X);
```

接着，我们追加IV的长度并把它和*H*相乘。结果就是GHASH的输出和初始*Y*值。

```
         /* copy counter out */
         XMEMCPY(gcm->Y, gcm->X, 16);
         zeromem(gcm->X, 16);
      } else {
         XMEMCPY(gcm->Y, gcm->buf, 12);
         gcm->Y[12] = 0;
         gcm->Y[13] = 0;
         gcm->Y[14] = 0;
         gcm->Y[15] = 1;
      }
```

这段代码处理IV是12个字节的情况（因为标记没有开启而且buflen为12）。在这种情况下，对IV的处理意味着简单地把它复制到GCM状态，并且把后32位设置为big-endian形式的“1”。

理想情况下，你可能想在GCM中使用12个字节的IV，因为它允许快速的包处理。如果你的GCM密钥在每次会话中都是随机衍生出来的，IV可以简单地作为一个包计数器。只要它们

都是惟一的，我们就可以合理地使用GCM。

```
        XMEMCPY(gcm->Y_0, gcm->Y, 16);
        zeromem(gcm->buf, 16);
        gcm->buflen = 0;
        gcm->totlen = 0;
        gcm->mode   = GCM_MODE_AAD;
     }

     if (gcm->mode != GCM_MODE_AAD || gcm->buflen >= 16) {
        return CRYPT_INVALID_ARG;
     }
```

这时，我们检查是否实际上是在AAD模式下，而且buflen是否是一个合法的值。

```
     x = 0;
#ifdef LTC_FAST
     if (gcm->buflen == 0) {
        for (x = 0; x < (adatalen & ~15); x += 16) {
            for (y = 0; y < 16; y += sizeof(LTC_FAST_TYPE)) {
                *((LTC_FAST_TYPE*)(&gcm->X[y])) ^=
                *((LTC_FAST_TYPE*)(&adata[x + y]));
            }
            gcm_mult_h(gcm, gcm->X);
            gcm->totlen += 128;
        }
        adata += x;
     }
#endif
```

如果设置了LTC_FAST，我们就用16个字节的增量来处理AAD数据，并直接快速地把它异或到GCM的状态中。这避免了所有的手动单字节异或操作。

```
     /* start adding AAD data to the state */
     for (; x < adatalen; x++) {
         gcm->X[gcm->buflen++] ^= *adata++;

         if (gcm->buflen == 16) {
           /* GF mult it */
           gcm_mult_h(gcm, gcm->X);
           gcm->buflen = 0;
           gcm->totlen += 128;
        }
     }
```

这是AAD数据的默认处理程序。当LTC_FAST定义时，它就能处理任何延留的字节。最好是让AAD数据总是在长度是16个字节的倍数。这样，我们就能够避免更慢的手动字节异或操作。

```
     return CRYPT_OK;
  }
```

7. GCM的明文处理

处理明文是处理一个GCM消息的最后一步，它在处理IV和AAD之后。由于我们是使用CTR来加密数据的，所以密文的长度和明文一样。

gcm_process.c:

```
  /**
```

```
   Process plaintext/ciphertext through GCM
   @param gcm       The GCM state
   @param pt        The plaintext
   @param ptlen     The plaintext length
   @param ct        The ciphertext
   @param direction Encrypt or Decrypt mode
   @return CRYPT_OK on success
  */
int gcm_process(    gcm_state *gcm,
                unsigned char *pt, unsigned long ptlen,
                unsigned char *ct,
                              int direction)
{
   unsigned long x, y;
   unsigned char b;
   int           err;

   LTC_ARGCHK(gcm != NULL);
   if (ptlen > 0) {
      LTC_ARGCHK(pt  != NULL);
      LTC_ARGCHK(ct  != NULL);
   }

   if (gcm->buflen > 16 || gcm->buflen < 0) {
      return CRYPT_INVALID_ARG;
   }

   if ((err = cipher_is_valid(gcm->cipher)) != CRYPT_OK) {
      return err;
   }
```

同样的错误检测代码。我们可以用一个空的明文来调用这个函数，就像我们用空的IV和AAD分组来调用前面两个函数一样。GCM对于长度为0的明文是有效的，并且也可以用来对AAD数据产生一个MAC标记。但这样一来使用GCM并不是一种好的方法，因为CMAC对于这个目的更加高效（而且也不需要大量的表来提升性能）。简而言之，如果你想要的只是一个MAC标记，那就别使用GCM。

```
   /* in AAD mode? */
   if (gcm->mode == GCM_MODE_AAD) {
      /* let's process the AAD */
      if (gcm->buflen) {
         gcm->totlen += gcm->buflen * CONST64(8);
         gcm_mult_h(gcm, gcm->X);
      }
```

这时，我们已结束了对AAD的处理。注意不像处理IV那样，我们没有结束GHASH。

```
      /* increment counter */
      for (y = 15; y >= 12; y--) {
          if (++gcm->Y[y] & 255) { break; }
      }
      /* encrypt the counter */
      if ((err =
         cipher_descriptor[gcm->cipher].ecb_encrypt(gcm->Y,
                                                     gcm->buf,
                                                     &gcm->K))
```

```
            != CRYPT_OK) {
            return err;
         }
```

我们递增Y的初始值并把它加密到buf数组中。这个缓冲区是用于加密或解密消息的CTR密钥。

```
         gcm->buflen = 0;
         gcm->mode   = GCM_MODE_TEXT;
      }

      if (gcm->mode != GCM_MODE_TEXT) {
         return CRYPT_INVALID_ARG;
      }
```

此时，我们已经准备好处理明文了。我们再次使用了LTC_FAST技巧来更加有效地处理明文。由于GHASH是用于密文的，所以我们必须不同地处理加密和解密。这也是因为我们允许用户提供的明文和密文缓冲区可以重叠。

```
      x = 0;
   #ifdef LTC_FAST
      if (gcm->buflen == 0) {
         if (direction == GCM_ENCRYPT) {
            for (x = 0; x < (ptlen & ~15); x += 16) {
```

我们再次试图一次处理16个字节的明文。由于这个原因，保证你的明文是16个字节的倍数是个不错的主意。

```
               /* ctr encrypt */
               for (y = 0; y < 16; y += sizeof(LTC_FAST_TYPE)) {
                   *((LTC_FAST_TYPE*)(&ct[x + y])) =
                      *((LTC_FAST_TYPE*)(&pt[x+y])) ^
                      *((LTC_FAST_TYPE*)(&gcm->buf[y]));
                  *((LTC_FAST_TYPE*)(&gcm->X[y])) ^=
                      *((LTC_FAST_TYPE*)(&ct[x+y]));
               }
```

这个循环把CTR密钥流和明文进行异或，然后把密文异或到GHASH累加器中。这个循环也许看起来很复杂，但GCC把这个循环很好地优化了。实际上，这个循环被完全地拆分并且转化为简单的加载和异或操作。

```
               /* GMAC it */
               gcm->pttotlen += 128;
               gcm_mult_h(gcm, gcm->X);
```

由于我们处理的是16个字节的分组，所以我们总是把累积的数据乘上H。

```
               /* increment counter */
               for (y = 15; y >= 12; y--) {
                   if (++gcm->Y[y] & 255) { break; }
               }
               if ((err =
               cipher_descriptor[gcm->cipher].ecb_encrypt(
                      gcm->Y, gcm->buf, &gcm->K)) != CRYPT_OK){
                  return err;
               }
```

接着我们递增CTR计数器，并对它进行加密以生成密钥流的其他16个字节。

```
            }
        } else {
           for (x = 0; x < (ptlen & ~15); x += 16) {
               /* ctr encrypt */
               for (y = 0; y < 16; y += sizeof(LTC_FAST_TYPE)) {
                   *((LTC_FAST_TYPE*)(&gcm->X[y])) ^=
                      *((LTC_FAST_TYPE*)(&ct[x+y]));
                   *((LTC_FAST_TYPE*)(&pt[x + y])) =
                      *((LTC_FAST_TYPE*)(&ct[x+y])) ^
                      *((LTC_FAST_TYPE*)(&gcm->buf[y]));
               }
               /* GMAC it */
               gcm->pttotlen += 128;
               gcm_mult_h(gcm, gcm->X);
               /* increment counter */
               for (y = 15; y >= 12; y--) {
                   if (++gcm->Y[y] & 255) { break; }
               }
               if ((err =
                    cipher_descriptor[gcm->cipher].ecb_encrypt(
                    gcm->Y, gcm->buf, &gcm->K)) != CRYPT_OK){
                  return err;
               }
           }
       }
    }
#endif
```

这段代码处理解密的情况。它类似于加密，但由于我们想修整时钟周期，所以没有把它们合并。代码大小的增加不重要，尤其是和节省的时间相比较的时候。

```
    /* process text */
    for (; x < ptlen; x++) {
```

这个循环对于加密和解密中的所有剩余字节进行处理。由于我们是一次一个字节地处理数据，最好在高性能的应用程序中使用这一节代码。

```
        if (gcm->buflen == 16) {
```

对于每16个字节，我们必须把它们累加到GHASH标记中并且修改CTR密钥流。

```
           gcm->pttotlen += 128;
           gcm_mult_h(gcm, gcm->X);

           /* increment counter */
           for (y = 15; y >= 12; y--) {
               if (++gcm->Y[y] & 255) { break; }
           }
           if ((err=cipher_descriptor[gcm->cipher].ecb_encrypt(
                gcm->Y, gcm->buf, &gcm->K)) != CRYPT_OK) {
              return err;
           }
           gcm->buflen = 0;
        }

        if (direction == GCM_ENCRYPT) {
           b = ct[x] = pt[x] ^ gcm->buf[gcm->buflen];
```

```
134         } else {
135            b = ct[x];
136            pt[x] = ct[x] ^ gcm->buf[gcm->buflen];
137         }
138         gcm->X[gcm->buflen++] ^= b;
```

这最后的一点代码看起来太过于复杂，但设计就是这么做的。我们允许ct=pt，这意味着如果不复制密文字节的话，就无法覆盖缓冲区。我们也可以把第138行代码移动到if语句的第二种情况中，但那样会扩大代码并没有节省时间。

```
139      }
140
141      return CRYPT_OK;
142   }
```

这时，我们已有足够的函数来开始一个GCM状态、添加IV、添加AAD和处理明文。现在我们必须结束GCM状态来计算最终的MAC标记了。

8. 结束GCM的状态

一旦已经处理完了GCM消息，我们就想计算GHASH的输出并得到MAC标记。

gcm_done.c:

```
/**
  Terminate a GCM stream
  @param gcm     The GCM state
  @param tag     [out] The destination for the MAC tag
  @param taglen  [in/out]  The length of the MAC tag
  @return CRYPT_OK on success
 */
int gcm_done(    gcm_state *gcm,
             unsigned char *tag, unsigned long *taglen)
{
   unsigned long x;
   int err;

   LTC_ARGCHK(gcm     != NULL);
   LTC_ARGCHK(tag     != NULL);
   LTC_ARGCHK(taglen  != NULL);

   if (gcm->buflen > 16 || gcm->buflen < 0) {
      return CRYPT_INVALID_ARG;
   }

   if ((err = cipher_is_valid(gcm->cipher)) != CRYPT_OK) {
      return err;
   }


   if (gcm->mode != GCM_MODE_TEXT) {
      return CRYPT_INVALID_ARG;
   }
```

这又是同样的程序健壮性检查。我们在所有的GCM函数这么做是很重要的，因为状态的上下文对于安全地使用GCM是很重要的。

```
   /* handle remaining ciphertext */
```

```
    if (gcm->buflen) {
       gcm->pttotlen += gcm->buflen * CONST64(8);
       gcm_mult_h(gcm, gcm->X);
    }

    /* length */
    STORE64H(gcm->totlen, gcm->buf);
    STORE64H(gcm->pttotlen, gcm->buf+8);
    for (x = 0; x < 16; x++) {
        gcm->X[x] ^= gcm->buf[x];
    }
    gcm_mult_h(gcm, gcm->X);
```

这结束了AAD和密文的GHASH。AAD的长度和密文的长度加到最后一个和H相乘的运算中。其输出是GHASH的最终值，但还不是MAC标记。

```
    /* encrypt original counter */
    if ((err =
         cipher_descriptor[gcm->cipher].ecb_encrypt(
             gcm->Y_0, gcm->buf, &gcm->K)) != CRYPT_OK) {
       return err;
    }
    for (x = 0; x < 16 && x < *taglen; x++) {
        tag[x] = gcm->buf[x] ^ gcm->X[x];
    }
    *taglen = x;
```

我们对原始的Y值进行加密，并把输出和GHASH的输出相异或。因为GCM允许截短MAC标记，所以我们就可以这样做。用户可以请求从0～16个字节的任意长度的MAC标记。但不建议截短GCM标记。

```

    cipher_descriptor[gcm->cipher].done(&gcm->K);

    return CRYPT_OK;
}
```

最后一个函数结束分组密码的状态。使用LibTomCrypt，这些分组密码可以在它们的初始化过程中分配资源，例如堆或者硬件令牌。这个函数要释放这些资源，如果有的话。当这个函数运行完成时，用户就得到了MAC标记并且GCM状态不再有效。

7.2.4 GCM的优化

既然我们已经看过了GCM的设计，以及一个可以应用于实际的来自LibTomCrypt工程的实现，我们就要特别地提一提怎样最好地使用GCM了。

LibTomCrypt中默认的实现，在定义了GCM_TABLES的情况下，每个GCM状态使用64KB的存储空间。这对于一台服务器或者桌面应用程序可能算不上什么问题，但是对于存储空间小于64KB的平台来说，它就不好使用了。有一种简单的折衷，它使用一个单独的8×128，4KB大小的表。实际上，它使用的是64KB变量中的第0个表，产生一个32个字节的乘积并使用和gcm_gf_mult()函数快速变形中一样的方法来约简它。

```
void gcm_mult_h(gcm_state *gcm, unsigned char *I)
{
   unsigned char T[32];
   int           i, j;

   /* produce 32 byte product */
   for (i = 0; i < 32; i++) T[i] = 0;
   for (i = 0; i < 16; i++)
      for (j = 0; j < 16; j++)
         T[i+j] ^= gcm->PC[I[i]][j];

   /* reduce it */
   for (i = 31; i >= 16; i--) {
     T[i-16] ^= gcm_shift_table[((unsigned)T[i]<<1)];
     T[i-15] ^= gcm_shift_table[((unsigned)T[i]<<1)+1];
   }

   /* copy out result */
   for (i = 0; i < 16; i++) I[i] = T[i];
}
```

我们可以使用面向字的异或操作来优化其中的嵌套循环。最后的约简不太需要优化。这个算法的退路是使用gcm_gf_mult()函数的LTC_FAST变量，它是通用GF（2）乘法函数，来执行乘以H的函数。

SIMD指令的使用

一些处理器具有被称为单指令多数据流（Single Instruction Multiple Data，SIMD）的指令系统，例如近年来的x86系列和基于PowerPC的G4和G5。这些指令允许开发者执行多路并行操作，例如128位宽的异或。这确实非常方便。例如，考虑支持SSE2的gcm_mult_h（）函数。

```
gcm_mult_h.c:  (From LibTomCrypt v1.14)
void gcm_mult_h(gcm_state *gcm, unsigned char *I)
{
   unsigned char T[16];
#ifdef GCM_TABLES
   int x, y;
#ifdef GCM_TABLES_SSE2
   asm("movdqa (%0),%%xmm0"::"r"(&gcm->PC[0][I[0]][0]));
   for (x = 1; x < 16; x++) {
      asm("pxor (%0),%%xmm0"::"r"(&gcm->PC[x][I[x]][0]));
   }
   asm("movdqa %%xmm0,(%0)"::"r"(&T));
<snip>
   XMEMCPY(I, T, 16);
}
#endif
```

正如我们所看到的，循环变得更加高效。特别地是，我们使用了更少的加载操作并且消耗了更少的解码器资源。例如，即使Opteron FPU在后台使用两个64位的操作来代替这些128位的操作，但由于我们只解码一个x86操作码，该处理器仍然能更有效地压缩这些指令。表7-2是在一个Opteron和Intel P4 Prescott处理器上对3种GCM实现的性能比较。

表7-2 GCM的实现性能观测

实　　现	每个消息字节（4KB个分组）	所用的时钟周期
LTC_FAST	69	
LTC_FAST和GCM_TABLES	27	53
LTC_FAST和GCM_TABLES_SSE2	25	49

7.2.5 CCM的设计

CCM是当前NIST认可的组合模式（SP 800-38C），它基于CBC-MAC和CTR链接模式。CCM的目的是使用一个单独的秘密密钥K和分组密码，对一个消息即能认证也能加密。不像GCM，CCM更加简单而且只需要分组密码来操作。由于这个原因，在使用GF（2）的乘法比较麻烦的情况下，它能更好地适应。

虽然CCM比GCM简单，但它确实有一些不足。在硬件环境中，GF（2）的乘法通常比一个AES加密（AES是经常和CGM配对的分组密码）要快。CCM也能处理AAD（它们称为报头数据），不过由于所用的操作顺序而使得它不太灵活。在GCM中你可以处理任意长度的AAD和明文数据，而CCM在初始状态之前就必须事先知道这些元素的长度。这意味着CCM不能用于数据流调用中。幸运地是，在实际应用中这通常不是一个问题。如果那些单独的数据包的长度是已知的或者可计算的，那么你只要简单地把CCM用在它们上面，所有的事情都会顺利进行。

CCM的设计可以分成3个不同的阶段。首先，我们必须生成一个B_0分组，它是由用户提供的nonce和消息长度来构建的。B_0分组用作CBC-MAC的IV和CTR链接模式的计数器。然后是执行MAC标记的生成，它是B_0、报头数据（AAD）和明文的CBC-MAC。和GCM不同的是，CCM会对明文进行认证，这是为什么一个惟一的nonce对于安全是很重要的原因。最后，加密明文，使用的是把一个修改了的B_0分组用做计数器的CTR链接模式。

1. CCM B_0的生成

B_0分组是CCM模式中的一个特殊的分组，它是消息的前缀（在报头数据之前），并衍生自用户提供的nonce值。B_0分组的形式取决于明文的长度，尤其是其长度编码的长度。例如，如果明文是129个字节长，将需要一个字节来对明文的长度进行编码。我们把这个长度称为q，明文的长度为Q。例如，在我们前一个例子中，有q=1且Q=129。

nonce数据可以一直到13个字节的长度，其内容用N来表示。需要的MAC标记的长度用t来表示，它必须是偶数且要大于2（如图7-4所示）。

八进制数	0	1 ... 15–q	16–q ...	15
内容	标志	N	Q	

图7-4 CCM B_0的格式

注意怎样截短nonce(N)取决于明文的长度。通常，在大部分的网络协议中这不是一个问题，其中Q通常小于65 536（这使得我们可以使用一个多达13个字节的nonce）。标志（flags）需要一个单独的字节而且用图7-5所示的来存储。

位数	7	6	5	4	3	2	1	0
内容	保留	Adata	$(t-2)/2$			$q-1$		

图7-5 CCM标志的格式

当出现报头数据（AAD）时，Adata标志就会被设置。我们通过把MAC标记的长度减去2然后再除以2来对它进行编码，而且t必须是集合{4, 6, 8, 10, 12, 14, 16}中的一个元素。让MAC的标记这么短通常不是一个好的主意。如果你确实想把每个包都削掉一些字节，试试使用一个不短于12个字节的MAC标记长度。明文的长度是通过减去1来编码的。q必须是集合{2, 3, 4, 5, 6, 7, 8}中的一个成员。对于（$t-2$）/2或者$q-1$的值来说，0都不是一个有效的值。

2. CCM MAC标记的生成

CCM的MAC标记是由B_0、所有的报头数据（如果出现的话）以及明文（如果有的话）的CBC-MAC所生成。我们在报头前加上它的长度。如果报头数据的长度小于65 280个字节，那么其长度就简单地以big-endian格式的两个字节值来编码。如果更长的话，就插入值0xFF FE，然后是报头长度的4个字节的big-endian格式的编码。从技术上说，CCM支持更大的报头长度，但如果你的报头数据多于4GB，就应该重新思考一下你的应用程序所使用的加密机制了。

报头数据通过在其尾部插入一些0字节来填充至16个字节的倍数。明文简单地按照所要求的进行填充。

在CBC-MAC的输出生成之后，它将先被存储在明文中，然后和明文一起以CTR模式进行加密。

3. CCM的加密

加密是以CTR模式来执行的，并使用B_0分组作为计数器。这里仅有的区别是，通过把标记和Q参数清零（只留下nonce）来修改B_0分组。对于标记和明文的每16个字节的分组，B_0分组的后q个字节将以big-endian的形式进行递增。密文和明文的长度相同，而且密文没有被填充。

7.2.6 CCM的实现

我们使用LibTomCrypt中的CCM实现作为参考。它是一个紧凑并高效的CCM实现，而且是在一个单独的函数调用中执行全部的CCM编码。由于它在一个单独的函数中接受所有的参数，所以它有许多参数。表7-3把实现中的名称和设计中的变量名称进行了匹配。

表7-3 CCM实现导引

设计名称	实现名称	功能
	cipher	所用的128位的分组密码在LTC表格中的索引，这使得CCM不知道分组密码的选择
K	key	秘密密钥
	keylen	密钥的长度，用八进制表示
	uskey	前面调度过的密钥，通过不使用密钥调度来节省时间
N	none	CCM的nonce
	noncelen	nonce的长度，用八进制表示

（续）

设计名称	实现名称	功　　能
	header	报头或者AAD数据
	headerlen	报头的长度，用八进制表示
P	pt	明文
Q	ptlen	明文的长度，用八进制表示
C	ct	密文
T	tag	MAC标记
t	taglen	需要的MAC标记的长度

ccm_memory.c:

```
/**
   CCM encrypt/decrypt and produce an authentication tag
   @param cipher     The index of the cipher desired
   @param key        The secret key to use
   @param keylen     The length of the secret key (octets)
   @param uskey      A previously scheduled key [can be NULL]
   @param nonce      The session nonce [use once]
   @param noncelen   The length of the nonce
   @param header     The header for the session
   @param headerlen  The length of the header (octets)
   @param pt         [out] The plaintext
   @param ptlen      The length of the plaintext (octets)
   @param ct         [out] The ciphertext
   @param tag        [out] The destination tag
   @param taglen     [in/out] The max size of the
                     authentication tag
   @param direction  Encrypt or Decrypt direction (0 or 1)
   @return CRYPT_OK if successful
*/
int ccm_memory(int cipher,
    const unsigned char *key,    unsigned long keylen,
    symmetric_key        *uskey,
    const unsigned char *nonce,  unsigned long noncelen,
    const unsigned char *header, unsigned long headerlen,
          unsigned char *pt,     unsigned long ptlen,
          unsigned char *ct,
          unsigned char *tag,    unsigned long *taglen,
                    int  direction)
{
   unsigned char  PAD[16], ctr[16], CTRPAD[16], b;
   symmetric_key *skey;
   int            err;
   unsigned long  len, L, x, y, z, CTRlen;

   if (uskey == NULL) {
      LTC_ARGCHK(key    != NULL);
   }
```

我们允许调用函数在调用之前调度一个密钥。这在实际应用中是一种不错的方法，因为它削掉了每次调用的相当数量的时钟周期。即使你没有使用LibTomCrypt，你还是应该在你的代码中提供或者使用这种优化。

```
   LTC_ARGCHK(nonce  != NULL);
```

```
    if (headerlen > 0) {
       LTC_ARGCHK(header != NULL);
    }
```

我们允许空的报头，因此在这些情况下，报头可以取NULL值。

```
    LTC_ARGCHK(pt     != NULL);
    LTC_ARGCHK(ct     != NULL);
    LTC_ARGCHK(tag    != NULL);
    LTC_ARGCHK(taglen != NULL);

#ifdef LTC_FAST
    if (16 % sizeof(LTC_FAST_TYPE)) {
       return CRYPT_INVALID_ARG;
    }
#endif
```

我们在这个实现的后面提供了一个面向字的优化。这个检查是一个简单的程序健壮性检查以确保它能实际运行。

```
     /* check cipher input */
    if ((err = cipher_is_valid(cipher)) != CRYPT_OK) {
       return err;
    }
    if (cipher_descriptor[cipher].block_length != 16) {
       return CRYPT_INVALID_CIPHER;
    }

    /* make sure the taglen is even and <= 16 */
    *taglen &= ~1;
    if (*taglen > 16) {
       *taglen = 16;
    }
```

我们强制标记长度为偶数并且如果它大于16个字节的话，就截短它。我们并不对过分长的标记长度报错，因为调用函数可能会简单地传入像如下这样的参数。

```
unsigned char tag[MAXTAGLEN];
unsigned long taglen;

taglen = sizeof(tag);
ccm_memory(..., &taglen, ...);
```

在这种情况下，如果它们支持多于一个的算法，那么MAXTAGLEN可以大于16。

```
    /* can't use < 4 */
    if (*taglen < 4) {
       return CRYPT_INVALID_ARG;
    }
```

按照CCM规范，我们必须拒绝标记小于4个字节的长度。我们没有选择只有报错，因为调用函数没有表示我们可以存储至少4个字节的MAC标记。

```
    /* is there an accelerator? */
    if (cipher_descriptor[cipher].accel_ccm_memory != NULL) {
        return cipher_descriptor[cipher].accel_ccm_memory(
            key,    keylen,
            uskey,
```

```
                nonce,   noncelen,
                header,  headerlen,
                pt,      ptlen,
                ct,
                tag,     taglen,
                direction);
      }
```

LibTomCrypt具有“重载”函数的能力——CCM就是如此。如果指针不为NULL，这个计算会自动地卸载它。用这种方法，开发者能利用加速器的优点而不用重写他们的应用程序。这种技术不是CCM的一部分，因此如果你需要的话，可以不用看这一大块代码。

```
      /* let's get the L value */
      len = ptlen;
      L   = 0;
      while (len) {
         ++L;
         len >>= 8;
      }
      if (L <= 1) {
         L = 2;
      }
```

这里我们计算L的值（即CCM设计中的q）。L是这个变量的原始名称，而且也是为什么我们在这里使用它的原因。按照CCM规范，我们要确保L至少为2。

```
      /* increase L to match the nonce len */
      noncelen = (noncelen > 13) ? 13 : noncelen;
      if ((15 - noncelen) > L) {
         L = 15 - noncelen;
      }
      if (15 < (noncelen + L)) noncelen = 15 - L;
```

如果nonce太大的话，就调整它的大小，同时根据需要调整参数L的大小。调用函数需要意识到这种折衷。例如，如果你想要加密1MB的包，你至少需要3个字节来对长度进行密码，这意味着nonce只能是12个字节长。你也可以增加一个检查来确保L从不会太小于明文的长度。

```
      /* allocate mem for the symmetric key */
      if (uskey == NULL) {
         skey = XMALLOC(sizeof(*skey));
         if (skey == NULL) {
            return CRYPT_MEM;
         }

         /* initialize the cipher */
         if ((err =
               cipher_descriptor[cipher].setup(
                  key, keylen, 0, skey)) != CRYPT_OK) {
            XFREE(skey);
            return err;
         }
      } else {
         skey = uskey;
      }
```

如果调用函数没有提供一个密钥，我们必须调度一个。我们通过从堆中分配调度密钥结构

来避免把它放到栈中。这对于嵌入式和内核应用程序很重要，因为栈会有相当的大小限制。

```
120     /* form B_0 == flags | Nonce N | l(m) */
121     x = 0;
122     PAD[x++] = ((headerlen > 0) ? (1<<6) : 0) |
123              (((*taglen - 2)>>1)<<3)       |
124              (L-1);
125
126     /* nonce */
127     for (y = 0; y < (16 - (L + 1)); y++) {
128         PAD[x++] = nonce[y];
129     }
130
131     /* store len */
132     len = ptlen;
133
134     /* shift len so the upper bytes of len are
135      * the contents of the length */
136     for (y = L; y < 4; y++) {
137         len <<= 8;
138     }
139
140     /* store l(m) (only store 32-bits) */
141     for (y = 0; L > 4 && (L-y)>4; y++) {
142         PAD[x++] = 0;
143     }
144     for (; y < L; y++) {
145         PAD[x++] = (len >> 24) & 255;
146         len <<= 8;
147     }
```

这段代码创建用于CCM的CBC-MAC阶段的B_0的值。数组PAD保存用于MAC的16个字节的CBC数据，而我们稍后会看到的CTRPAD保存了16个字节的CTR输出。

分组的第一个字节（第122行代码）是标志。我们根据*headerlen*来设置*Adata*标志，通过把*taglen*除以2来对标记长度进行编码，并且最后保存明文的长度。

接着，nonce被复制到分组中。我们只使用nonce的16–L+1个字节，因为我们必须保存标志和*L*个字节的明文长度值。

为了让它更加实用，我们只存储明文长度的32位。如果用户指定了一个短的nonce，*L*的值必须增加以补充。在本例中，在对实际长度进行编码之前我们用一些0字节来填充。

```
149     /* encrypt PAD */
150     if ((err =
151          cipher_descriptor[cipher].ecb_encrypt(
152             PAD, PAD, skey)) != CRYPT_OK) {
153         goto error;
154     }
```

我们通过一个全0的IV来有效地使用CBC-MAC，因此我们必须做的第一件事就是对PAD进行加密。现在，密文就是剩下的报头和明文数据的CBC-MAC的IV。

```
156     /* handle header */
157     if (headerlen > 0) {
158        x = 0;
```

我们仅到这里并且如果有要处理的报头数据的话，就执行下面的代码。

```
      /* store length */
      if (headerlen < ((1UL<<16) - (1UL<<8))) {
         PAD[x++] ^= (headerlen>>8) & 255;
         PAD[x++] ^= headerlen & 255;
      } else {
         PAD[x++] ^= 0xFF;
         PAD[x++] ^= 0xFE;
         PAD[x++] ^= (headerlen>>24) & 255;
         PAD[x++] ^= (headerlen>>16) & 255;
         PAD[x++] ^= (headerlen>>8) & 255;
         PAD[x++] ^= headerlen & 255;
      }
```

报头数据长度的编码取决于报头数据的大小。如果它小于65 280个字节，我们使用短的两字节编码。否则，我们使用换码顺序0xFF FE，然后是4字节编码。CCM支持更大的报头大小，但你不可能永远需要支持它。

注意我们不是把PAD（作为一个IV）和其他缓冲区异或，而是简单地把长度异或到PAD中。这避免了我们可能会另外使用任何的双缓冲。

```
      /* now add the data */
      for (y = 0; y < headerlen; y++) {
          if (x == 16) {
             /* full block so let's encrypt it */
             if ((err =
                  cipher_descriptor[cipher].ecb_encrypt(
                     PAD, PAD, skey)) != CRYPT_OK) {
               goto error;
             }
             x = 0;
          }
          PAD[x++] ^= header[y];
      }
```

这个循环处理整个报头数据。我们没有提供任何一种LTC_FAST优化，因为报头一般为空或者非常短。对于每16个报头字节，我们对PAD进行加密以合理的模拟CBC-MAC。

```
      /* remainder? */
      if (x != 0) {
         if ((err =
              cipher_descriptor[cipher].ecb_encrypt(
                 PAD, PAD, skey)) != CRYPT_OK) {
            goto error;
         }
      }
   }
```

如果还有剩下的报头数据（即headerlen不是16的倍数），那么就用0字节对它进行填充，然后对它加密。由于和0字节异或是一种无操作，所以我们简单地忽略这一步并直接调用分组密码。

```
   /* setup the ctr counter */
   x = 0;

   /* flags */
   ctr[x++] = L-1;

```

```
      /* nonce */
      for (y = 0; y < (16 - (L+1)); ++y) {
         ctr[x++] = nonce[y];
      }
      /* offset */
      while (x < 16) {
         ctr[x++] = 0;
      }
```

这个代码为CTR加密模式创建初始计数器。标志只包含明文长度的长度。nonce像用于CBC-MAC时一样进行复制，分组其余的部分都被清零。在加密过程中，nonce后的字节会递增。

```
      x      = 0;
      CTRlen = 16;

      /* now handle the PT */
      if (ptlen > 0) {
         y = 0;
#ifdef LTC_FAST
         if (ptlen & ~15)  {
             if (direction == CCM_ENCRYPT) {
                for (; y < (ptlen & ~15); y += 16) {
```

如果是在加密，我们就处理明文全部的16字节分组。

```
                   /* increment the ctr? */
                   for (z = 15; z > 15-L; z--) {
                       ctr[z] = (ctr[z] + 1) & 255;
                       if (ctr[z]) break;
                   }
```

CTR计数器是big-endian而且是存储在数组ctr的末尾。这段代码对它进行加1递增。

```
                   if ((err =
                        cipher_descriptor[cipher].ecb_encrypt(
                           ctr, CTRPAD, skey)) != CRYPT_OK) {
                      goto error;
                   }
```

在使用CTR计数器对明文进行加密之前，我们必须对它进行加密。

```
                   /* xor the PT against the pad first */
                   for (z = 0; z < 16; z += sizeof(LTC_FAST_TYPE)) {
                       *((LTC_FAST_TYPE*)(&PAD[z]))  ^=
                          *((LTC_FAST_TYPE*)(&pt[y+z]));
                       *((LTC_FAST_TYPE*)(&ct[y+z])) =
                          *((LTC_FAST_TYPE*)(&pt[y+z])) ^
                          *((LTC_FAST_TYPE*)(&CTRPAD[z]));
                   }
```

这个循环把明文的16个字节和CBC-MAC的pad进行异或，然后通过把CTRPAD和明文异或创建16个字节的密文。我们在后面进行加密（在CBC-MAC之后），因为我们允许明文和密文指向同一个缓冲区。

```
                   if ((err =
                        cipher_descriptor[cipher].ecb_encrypt(
                           PAD, PAD, skey)) != CRYPT_OK) {
```

```
                    goto error;
                 }
              }
```

对CBC-MAC pad的加密执行了所需的对明文16字节分组的MAC操作。

```
           } else {
              for (; y < (ptlen & ~15); y += 16) {
                 /* increment the ctr? */
                 for (z = 15; z > 15-L; z--) {
                     ctr[z] = (ctr[z] + 1) & 255;
                     if (ctr[z]) break;
                 }
                 if ((err =
                      cipher_descriptor[cipher].ecb_encrypt(
                         ctr, CTRPAD, skey)) != CRYPT_OK) {
                    goto error;
                 }

                 /* xor the PT against the pad last */
                 for (z = 0; z < 16; z += sizeof(LTC_FAST_TYPE)) {
                     *((LTC_FAST_TYPE*)(&pt[y+z])) =
                        *((LTC_FAST_TYPE*)(&ct[y+z])) ^
                        *((LTC_FAST_TYPE*)(&CTRPAD[z]));
                     *((LTC_FAST_TYPE*)(&PAD[z]))  ^=
                        *((LTC_FAST_TYPE*)(&pt[y+z]));
                 }
                 if ((err =
                      cipher_descriptor[cipher].ecb_encrypt(
                         PAD, PAD, skey)) != CRYPT_OK) {
                    goto error;
                 }
              }
           }
        }
```

对解密的处理和加密类似，但也有所不同，因为我们允许明文和密文指向同一块内存。因为这段代码可以拆分，所以我们在主循环中的可能位置避免使用了冗余的条件代码。

```
#endif

        for (; y < ptlen; y++) {
            /* increment the ctr? */
            if (CTRlen == 16) {
               for (z = 15; z > 15-L; z--) {
                   ctr[z] = (ctr[z] + 1) & 255;
                   if (ctr[z]) break;
               }
               if ((err =
                    cipher_descriptor[cipher].ecb_encrypt(
                       ctr, CTRPAD, skey)) != CRYPT_OK) {
                  goto error;
               }
               CTRlen = 0;
            }

            /* if we encrypt we add the bytes to the MAC first */
            if (direction == CCM_ENCRYPT) {
```

```
                b      = pt[y];
                ct[y] = b ^ CTRPAD[CTRlen++];
             } else {
                b      = ct[y] ^ CTRPAD[CTRlen++];
                pt[y] = b;
             }

             if (x == 16) {
                if ((err =
                     cipher_descriptor[cipher].ecb_encrypt(
                        PAD, PAD, skey)) != CRYPT_OK) {
                   goto error;
                }
                x = 0;
             }
             PAD[x++] ^= b;
         }
```

这段代码执行了对所有没有被LTC_FAST代码处理的明文字节的CCM操作。之所以执行这段代码是因为明文不是16个字节的倍数，或者LTC_FAST没有定义。理想情况下，我们希望避免需要这段代码，因为它很慢而且在处理过许多包之后会消耗相当数量的处理能源。

```
         if (x != 0) {
            if ((err =
                 cipher_descriptor[cipher].ecb_encrypt(
                    PAD, PAD, skey)) != CRYPT_OK) {
              goto error;
            }
         }
      }
```

如果有字节剩下的话，我们就结束CBC-MAC。正如对报头的处理一样，我们隐式地用0字节对PAD进行填充并对它进行加密。此时，PAD包含了CBC-MAC值，但不是CCM标记，因为我们还需要对它加密。

```
      /* setup CTR for the TAG */
      for (z=15;z>15-L;z++) ctr[z] = 0x00;
      if ((err =
           cipher_descriptor[cipher].ecb_encrypt(
              ctr, CTRPAD, skey)) != CRYPT_OK) {
         goto error;
      }
```

用于CBC-MAC标记的CTR pad通过这样的方法来计算，首先把CTR计数器的后L个字节清零，然后把它加密，输出保存在CTRPAD中。

```
      if (skey != uskey) {
         cipher_descriptor[cipher].done(skey);
      }
```

如果调度我们自己的密钥，那么现在就可以释放所有分配的资源。

```
      /* store the TAG */
      for (x = 0; x < 16 && x < *taglen; x++) {
          tag[x] = PAD[x] ^ CTRPAD[x];
      }
```

```
    *taglen = x;
```

CCM允许可变长度的标记，长度从4～16个字节，步长为2个字节。我们通过把CBC-MAC标记和最后一个加密了的CTR计数器异或来加密和存储CCM标记。

```
#ifdef LTC_CLEAN_STACK
    zeromem(skey,   sizeof(*skey));
    zeromem(PAD,    sizeof(PAD));
    zeromem(CTRPAD, sizeof(CTRPAD));
#endif
```

这段代码把认为是敏感栈中的存储空间清零。我们希望栈中的数据还没有被交换到磁盘上，但这个程序并不做这种保证。通过把存储空间清零，任何潜在的栈泄露都不会把密钥或者CBC-MAC中间值和攻击者分享。我们只在用户需要并定义了LTC_CLEAN_STACK宏的情况下才执行这个操作。

提示 在大多数的现在操作系统中，一个程序（或进程）所使用的内存叫做虚拟内存。它没有固定的物理地址，可以移动并且甚至可以交换到磁盘上（通过使页面无效来实现）。后一种行为通常叫做交换内存，因为它允许用户模拟拥有比他们实际上所拥有的更多的物理内存。

但是，交换内存的缺点是进程内存可能包含敏感的信息，例如私钥、用户名、口令和其他证书。为了防止这种情况发生，应用程序可以锁住内存。在操作系统中，例如那些基于NT内核的（例如Win2K、WinXP），锁定是自动的而且操作系统能够选择稍后再把非内核数据交换出去。

在POSIX兼容的操作系统中，例如那些基于Linux和BSD内核的，提供了一系列使锁定更为容易的函数，例如mlock()、munlock()、mlockall()，等等。大多数系统中的内存都是珍贵的，因此优雅而合理的应用程序会要求锁定尽可能少的内存。大部分情况下，被锁定的内存会跨越一个包含几页内存的区域。在x86系列处理器上，一个页为4KB。这意味着所有锁定的内存实际上是4KB的数位。

理想情况下，一个应用程序应该把和它相关的证书集中起来以减少在内存中需要锁定它们的物理页数。

```
error:
    if (skey != uskey) {
       XFREE(skey);
    }

    return err;
}
```

一旦成功地完成这个函数，用户就得到了密文（或者明文，这取决于方向）以及CCM标记。虽然这个函数稍微有点长，但它却很好地把所有的操作都绑定在一个单独的函数调用中，这使得它的使用相当简单。

7.3 总结

本章介绍了两个分别由NIST和IEEE指定的标准的加密和认证模式。它们都是设计用来取

一个单独的密钥和IV（nonce）并且产生一个密文和消息认证码标记，从而以减少开发者必须支持的不同标准的数量来简化他们的处理，并且在实际应用中也减少了他们需要调用来达到同样结果的函数数量。

知道怎样使用这些模式就是知道怎样合理地选择一个IV，最佳地使用额外的认证数据（AAD），并且检查它们所产生的MAC标记。这两个模式都没有为开发者管理这些性质，所以开发者必须仔细地考虑它们。

对于大多数的应用程序来说，非常建议使用这两种模式，而不要使用一个特别的加密和认证的组合，如果不是主要因为代码简单的原因，那么也是因为合理地坚持密码学标准的原因。

7.3.1 这些模式可以用来做哪些事

在前一章中可以看到我们是怎样实现数据保密性和真实性的，是通过组合使用一个对称密码和带有一个MAC算法的链接模式。这里，这些模式的目标是把这两者组合到一起。这同时也实现了几种关键的目标。正如我们在前一章中间接提到的，CCM和GCM也可以用于小的包消息，对于保证双方之间数据流消息的安全是很理想的。CCM和GCM也可以用于离线任务，例如文件加密，但它们并不是为这些任务而设计的（尤其是CCM，因为它需要事先知道明文的长度）。

首先，把这些模式组合起来使得开发更为容易——只需要知道一个密钥和一个IV。这些模式会使用密钥和IV来处理加密和认证这两种任务。这也使得密钥衍生更简单更快，因为更少的会话数据需要衍生。这也意味着有更少的变量需要维护。

这些组合的模式也意味着在一个单独的函数调用中执行这两种目标是可能的。在我们尤其需要捕捉错误代码（通常是看返回代码）的代码中，调用更少的函数则意味着安全地写这些代码更为容易。虽然也有其他的方法来捕捉错误，例如信号和内部掩码，但这会使在C语言中线程安全的全局错误检测变得相当困难。

除了让代码更容易阅读和书写之外，这些组合的模式也使得安全分析更为简单。例如，CCM是CBC-MAC和CTR加密模式的组合。我们可以用各种办法把CCM的安全性简化到这些模式的安全性。一般来说，在有一个完整长度的MAC标记的情况下，CCM的安全性会简化到其分组密码的安全性（假设使用一个惟一的nonce和随机密钥）。

我们说简化的意思是，我们能够举出一个可以证明等价的论据。例如，如果CTR的安全性可以简化到分组密码的安全性的话，我们说它和后者同样安全。由于这个原因，如果一个人能够破解这个分组密码，那么他也能够破解CTR模式（严格来说，CTR的安全性可以简化到什么程序取决于这个分组密码是否是一个PRP）。

因此，在这种环境下，如果我们说CCM在作为一个合理的伪随机置换（PRP）的时候可以简化到分组密码的安全性，那么如果我们能够破解分组密码（通过证明它不是一个PRP），也就有可能攻破CCM。类似地，GCM在保密性方面可以简化到分组密码的安全性，在MAC方面可以简化到通用散列。证明它能够安全太复杂了。

7.3.2 选择一个Nonce

CCM和GCM都需要一个惟一的Nonce（仅用一次的N）值来维持它们的保密性和真实性目

标。在这两种情况下，其值并不需要是随机的，但只要求对一个给定的密钥，它应是惟一的。也就是说，你可以安全地在两个不同的密钥之间使用同样的nonce（但只能用一次）。一旦对一个特定的密钥使用这个nonce，那么你就不能再次使用它。

1. GCM 的Nonce

GCM的设计对于12个字节的nonce值会更加有效。不管是更长还是更短，GHASH都会用来为这个模式创建一个IV。在这种情况下，我们可以简单地使用12个字节的nonce作为一个包计数器。但是由于我们还要把nonce发送给其他的成员，我们可以因为这种应用目的而使用同一个nonce。每个包都会有它自己的12个字节的nonce（通过递增来实现），并且接收者可以通过检查nonce是否为一个96位的数来检测出重放以及乱序的包。

你可以把这个12个字节的数用作一个big-endian或者little-endia值，因为GCM不会截短nonce。

2. CCM的Nonce

CCM的设计并不像GCM那样偏好于nonce。如果你知道所有的包都将小于65 536个字节，那么你可以安全地假设nonce是允许使用13个字节的。像GCM一样，你可以把它用作一个104位的计数器并针对每个发送出去的包都把它递增。

如果你不能提前确定数据包的长度，最好默认使用一个更短的nonce（即11个字节，它可以允许多达4GB的包）作为计数器。记住，nonce的长度并没有什么神奇的性质，只是你不得不拥有一个足够长的nonce，使得你在相同的密钥下所发送的每个包都有一个惟一的值。

如果数据包太长（而没有空间来存储其长度）的话，CCM会截短nonce，因此在实际应用中最好把它看做是一个little-endian的计数器。这样最高字节会被截掉。只用一个更短的nonce会比担心它要好得多。

7.3.3 额外的认证数据

CCM和GCM都支持一种叫做额外的认证数据（AAD）的侧信道。这个数据是非私有数据而且应该能影响MAC标记的输出。也就是说，如果明文和AAD并不是同时出现而且没有被修改的话，标记应该能反应出这种情况。

AAD有用的地方是把会话元数据和包一起保存。常见的如用户名、会话ID和事务ID。你可能从不使用一个用户证书，因为在一个基于每个单独包的情况下，它可能并不是你真正需要的。

这两种协议都支持空的AAD串。只有GCM会对是16个字节长的倍数的AAD串进行优化处理。CCM插入一个4字节或者6字节的报头来补充数据并且使得优化更为困难。一般来说，如果你使用的是CCM，试着使用短的AAD串，最好小于10个字节，这样你就可以把它放到一个单独的加密调用中。对于GCM，试着让你的AAD串是16个字节的倍数，即使你不得不用0字节来填充（其实现不会为你做这件事，因为它会改变AAD的含意）。

7.3.4 MAC标记数据

由这两种实现所产生的MAC标记并不是在内部检查的。对它的典型用法是把MAC标记和

密文一起传输，并且接受者会用在解密消息时生成的标记和它进行比较。

至少从理论上来看，你可以截短MAC标记到短的长度，例如80位或者96位。但是，一些证据指出对GCM这是不可以的，而且在实际中这种节省是很小的。随着研究的进展，最好是阅读最新的对GCM和CCM的分析文章来看短的标记在实际应用中是否还是安全的。

在实际应用中，如果你通过一条稳定的信道来聚焦包的话，可以节省更多的空间。例如，发送196个字节或者256个字节的包，而不是发送128个字节的包。你能发送更少的nonce（以及协议数据），并且这种差别使得你能够使用一个更长的MAC标记。显然，这并不适用于低延迟交换的情况（例如VoIP），所以它并不是一个“防弹的”建议。

7.3.5 构造举例

至于我们的例子，我们将借助于第6章中的例子，不同的是不是使用HMAC和CTR模式，而是使用CCM。演示程序的其余部分都一样。我们将再次使用LibTomCrypt，因为它提供了一个不错的CCM接口，可以很容易地用于实际应用。

encmac.c:

```
#include <tomcrypt.h>

#define KEYLEN     16
```

我们只有一个密钥，其长度为16个字节。

```
/* Our Per Packet Sizes, Nonce len and MAC len */
#define NONCELEN      4
#define MACLEN        12
#define OVERHEAD      (NONCELEN+MACLEN)
```

和前面的例子一样，我们有一个包计数器长度（nonce的长度）、MAC标记长度和组合开销。这里我们有一个32位的计数器和一个96位的MAC。

```
/* errors */
#define MAC_FAILED    -3
#define PKTCTR_FAILED -4

/* our nice containers */
typedef struct {
   unsigned char PktCTR[NONCELEN],
                 key[KEYLEN];
   symmetric_key skey;
} encauth_channel;
```

我们的加密和认证信道和前一个例子很相似。区别在于我们使用的是一个通用调度的密钥，而且没有一个MAC密钥。

```
typedef struct {
   encauth_channel channels[2];
} encauth_stream;


void register_algorithms(void)
{
   register_cipher(&aes_desc);
```

```
   register_hash(&sha256_desc);
}

int init_stream(const unsigned char *masterkey,
                              unsigned  masterkeylen,
                   const unsigned char *salt,
                              unsigned  saltlen,
                         encauth_stream *stream,
                                   int  node)
{
   unsigned char tmp[2*KEYLEN];
   unsigned long tmplen;
   int           err;
   encauth_channel tmpswap;

   /* derive keys */
   tmplen = sizeof(tmp);
   if ((err = pkcs_5_alg2(masterkey, masterkeylen,
                          salt, saltlen,
                          16, find_hash("sha256"),
                          tmp, &tmplen)) != CRYPT_OK) {
      return err;
   }

   /* copy keys */
   memcpy(stream->channels[0].key,
          tmp, KEYLEN);
   memcpy(stream->channels[1].key,
          tmp + KEYLEN, KEYLEN);

   /* reset counters */
   memset(stream->channels[0].PktCTR, 0,
          sizeof(stream->channels[0].PktCTR));
   memset(stream->channels[1].PktCTR, 0,
          sizeof(stream->channels[1].PktCTR));

   /* schedule keys+setup mode */
   if ((err = rijndael_setup(stream->channels[0].key,
                             KEYLEN, 0,
                             &stream->channels[0].skey))
      != CRYPT_OK) {
      return err;
   }

   if ((err = rijndael_setup(stream->channels[1].key,
                             KEYLEN, 0,
                             &stream->channels[1].skey))
      != CRYPT_OK) {
      return err;
   }

   /* do we swap? */
   if (node != 0) {
       tmpswap            = stream->channels[0];
       stream->channels[0] = stream->channels[1];
       stream->channels[1] = tmpswap;
       zeromem(&tmpswap, sizeof(tmpswap));
   }
```

```

    zeromem(tmp, sizeof(tmp));

    return 0;
}
```

这个初始化函数实质上和前一个例子是一样的。我们只改变了要生成多少PKCS #5数据。

```
int encode_frame(const unsigned char *in,
                                  unsigned  inlen,
                             unsigned char *out,
                            encauth_stream *stream)
{
    int            x, err;
    unsigned long taglen;

    /* increment counter */
    for (x = NONCELEN-1; x >= 0; x--) {
        if (++(stream->channels[0].PktCTR[x]) & 255) break;
    }

    /* store counter */
    for (x = 0; x < NONCELEN; x++) {
      out[x] = stream->channels[0].PktCTR[x];
    }
```

我们使用包计数器PktCTR来作为一种保持包有序的方法。在CCM算法中，我们也把它用作一个nonce。生成的输出由3部分组成，首先是nonce，接着是密文，然后是MAC标记。

```
    /* encrypt it */
    taglen = MACLEN;
    if ((err = ccm_memory(
```

这个函数会给我们生成密文和MAC标记。这个算法库真的不错。

```
                  find_cipher("aes"),
```

首先，我们需要找到要使用的分组密码的索引。本例中的CCM使用的是AES。我们也可以使用其他的128位分组密码，例如Twofish、Anubis或者Serpent。

```
                  stream->channels[0].key,     KEYLEN,
```

这是我们要用的密钥字节数组和密钥长度。

```
                  &stream->channels[0].skey,
```

这是预调度过了的密钥。我们的CCM程序会使用它，而不是动态地调度密钥。为完整起见，我们也传递了这个字节数组（加速器可能会需要它）。

```
                  stream->channels[0].PktCTR, NONCELEN,
```

这是用于数据包的nonce。

```
                  NULL, 0,
```

这是AAD。本例中，我们不使用它。我们传递NULL和0来告诉算法库。

```
                  (unsigned char*)in, inlen, out + NONCELEN,
                  out+inlen+NONCELEN, &taglen, CCM_ENCRYPT))
```

这些是明文和密文缓冲区。总是首先指定明文，而不管我们所用的方向。对于那些期望一

个“输入”和“输出”参数顺序的读者来说，这可能会有一点让人感到困惑。

```
         != CRYPT_OK) {
        return err;
     }
     return CRYPT_OK;
  }
```

这个ccm_memory()函数调用产生密文以及MAC标记。包的格式包含nonce，接着是密文，然后是MAC标记。从安全的角度来看，这个函数提供了和前一章的这个函数的化身相同的安全目标。本例中的改进是我们缩短了代码并从结构中删除了一些变量。

```
int decode_frame(const unsigned char *in,
                             unsigned  inlen,
                        unsigned char *out,
                       encauth_stream *stream)
{
   int           err;
   unsigned char tag[MACLEN];
   unsigned long taglen;

   /* restore our original inlen */
   if (inlen < MACLEN+NONCELEN) { return -1; }
   inlen -= MACLEN+NONCELEN;
```

和以前一样，我们假设*inlen*是整个包的长度而仅仅是明文。我们首先确定它是一个有效的长度，然后从这个长度中减去MAC标记和nonce。

```
   /* decrypt it */
   taglen = sizeof(tag);
   if ((err = ccm_memory(
                 find_cipher("aes"),
                 stream->channels[1].key, KEYLEN,
                 &stream->channels[1].skey,
                 (unsigned char*)in, NONCELEN,
                 NULL, 0,
                 out, inlen, (unsigned char*)in + NONCELEN,
                 tag, &taglen, CCM_DECRYPT))
       != CRYPT_OK) {
      return CRYPT_OK;
   }
```

加密和解密都使用了ccm_memory()函数。在解密模式中，它和加密模式中的使用差不多，不同的是*out*缓冲区放在了明文的参数位置。因此，它出现在输入之前，这看起来有点不舒服。

注意，我们把MAC标记保存在局部变量中，因为后面我们需要对它进行比较。

```
   /* compare */
   if (memcmp(tag, in+inlen+NONCELEN, MACLEN)) {
      return MAC_FAILED;
   }

   /* compare CTR */
   if (memcmp(in, stream->channels[1].PktCTR, NONCELEN) <= 0) {
      return PKTCTR_FAILED;
   }
```

这些检查和前一个例子中的一样。第一个是避免修改，第二个是避免重放攻击。我们还没有处理乱序的包（例如使用一个窗口），但那应该很容易实现。

```
    /* copy the pktctr */
    memcpy(stream->channels[1].PktCTR, in, NONCELEN);
```

这时，我们的包是有效的，所以我们把nonce复制出来并保存它，这样就防止了重放攻击。

```
    return CRYPT_OK;
}

<snip>
```

演示程序剩下的部分和前一个例子一样，因为我们没有改变函数接口。

7.4 常见问题

下面的常见问题，由本书的作者所回答，它们即可以用来测试你对本章所出现的概念的理解，也可以帮助你在现实生活中实现这些概念。如果希望作者解答你的问题，请浏览www.syngress.com/solutions，然后点击Ask the Author表单。

问：什么是加密和认证模式？

答：这些是设计用来接受一个单独的密钥和IV，并且允许调用者对他的消息即可以加密也可以认证的算法。不像明显的加密和认证协议，这些模式不需要两个互相独立的密钥来保证安全。

问：这些模式对于明显的模式有什么好处？

答：对于这些模式实际上有两个好处。首先，它们可以很容易地融入到加密系统中，因为它们需要更少的密钥，并且同时执行两个功能。这意味着开发者能够用一个单独的函数调用来实现两个目标。这也意味着他们更有可能开始认证传输的数据。第二个好处是他们常常可以简化到底层的分组密码或者原型的安全性。

问：可简化是什么意思？

答：当我们说一个问题可以简化成另一个时，意思是说解决前者就等价于解决了后者。例如，CTR-AES的安全性可简化到AES是不是一个真正的伪随机置换（PRP）问题。如果AES不是一个PRP，那么CTR-AES就是不安全的。

问：有哪些标准化的加密和认证模式？

答：在NIST的公告SP 800-38C中指定了CCM。GCM是一个IEEE的标准而且以后将会在NIST的公告SP 800-38D中指定。

问：我应该使用这些模式来代替哪些明显的模式，例如AES-CTR和SHA1-HMAC吗？

答：这取决于你是否想遵从一个旧的标准。如果你不想，那么很可能GCM或者CCM会更好地利用系统资源来实现同样的目标。那些明显的模式在实际应用中只是安全的，但一般实现起来更加困难，并且在实际的应用程序中也得到了验证。它们也依赖于更安全的假设，这会造成安全风险。

问：什么是一个nonce，它的重要性是什么？

答：nonce是一个参数（类似于一个初始值），它只准备使用一次。Nonce用于把熵引入到

一个静态的而且是完全确定的进程中（即在选择完密钥之后）。在一个密钥下Nonce必须是惟一的，但对多个不同的密钥，它也可以重用。通常来讲，它们是简单的包计数器。如果没有一个惟一的nonce，GCM和CCM都没有可证明安全的性质。

问：什么是额外的认证数据？AAD？报头数据？

答：额外的认证数据（AAD），也叫做报头数据和关联数据（associated data）（在EAX模式中），是和一个消息相关的数据，它必须是MAC标记计算的一部分但不加密。通常，它是简单的和会话相关的元数据，它不是私有的但会影响对消息的解释。例如，一个IP数据报编码器可以使用IP数据报的一部分来作为报头数据。以这种方式，如果路由改变了，它也是可以检测出来的，并且由于IP头没有加密（不使用另一种IP标准），它必须仍然是明文的形式。这只是AAD的一个例子，还有其他的一些。在实际应用中，它并不经常使用，但标准还是提供了它。

问：我应该使用哪种模式？GCM还是CCM？

答：这取决于你所遵从的标准（如果有的话）。目前，CCM是这两者中惟一的NIST标准。如果你的应用环境是无线标准，你也许更需要GCM。除了这两种情况之外，它还取决于所用的平台。GCM是非常快的，但需要一个很大的64KB的表来达到这种速度（至少在软件环境中）。CCM稍微有点慢，但在设计和实现上都更为简单。CCM也需要一些很大的表来达到它的性能要求。如果你从GCM中删除这些表，它可能比CCM要慢许多。

问：这些模式有什么专利吗？

答：GCM和CCM都不是专利的。它们可以自由地用于各种目的。

问：我在哪里可以找到它们的实现？

答：目前，仅有LibTomCrypt以一个算法库的形式提供了GCM和CCM。Brian Gladman也有一个单独的有版权的实现。Crypto++一个也没有，但其作者已经意识到了这一点。

问：我可以用MAC标记做什么？

答：如果你是在加密，把它和你的密文以及AAD数据（如果有的话）一起传输出去。如果你是在解密，解密过程会生成一个MAC标记，把它与和密文一起存储的值进行比较。如果它们不相等，那么这个消息就是已经被修改了的或者存储的MAC标记是无效的。

第 8 章

大整数算术

本章解决方案：

- 什么是大数？
- 为什么密码学需要它们
- 什么算法是最重要的
- 实现细节
- 从哪里得到更多的资源

☑ 总结
☑ 快速查找解决方案
☑ 常见问题

8.1 简介

到目前为止，我们已经研究了对称密钥算法，它们主要依靠秘密密钥来获得安全性。现在我们即将探索公钥密钥学领域，但在这之前，需要介绍一个相当重要的数学知识。

大多数公钥算法一般都是基于很难解决的问题。例如，RSA算法（不严格来说）和大整数分解一样安全是困难的。即如果大整数分解是困难的，那么攻破RSA也是困难的（在实际应用中）。类似地，攻破椭圆曲线算法和攻破给定的曲线上的点乘逆运算也是同样困难的。

在这两种情况中，当你增加问题的规模时，“问题”会变得更加复杂。在RSA的例子中，当你增大其合数时（公钥），其分解会变得更加困难。类似地，当你增加椭圆曲线的阶（如果这时不知道它是什么意思，不用担心），点乘逆运算的困难性也会增加。

为了提供这些更大的参数，我们必须使用统称为BigNum算法的算法。

8.2 什么是BigNum

作为开发者，你很可能已经意识到了你所喜欢的数据类型的大小限制。例如，在C语言中，*int*数据类型在可移植的情形下，最大可以表示32 767。即使在某些平台上它也许是32位或者64位的长度，但你不能总是依靠这一点。

大部分程序设计语言在它们的设置中，最多有一种64位的数据类型。如果我们不得不直接在这些约束下生存，那么我们的公钥算法会非常不安全。例如，分解64位的数字是一种很简单的任务。

为了解决这些问题，我们使用通常叫做多（multiple）精度或者固定（fixed）精度的算法。这些算法构造了大整数的表示，通过使用已支持的数据类型（例如*unsigned long*）作为一个数字（digit）——很像我们以十进制的形式在0～9的范围内追加更多的数字来构造大整数一样。

这些数字（在不同的库中也有叫做零件（limb）的）形成了所有运算的基础。在目标平台上，它们的选择通常是高效的。例如，如果你的机器能够高效地执行*unsigned long*数据类型的操作，那么它很可能是你的数字类型。我们所要执行的数学计算非常类似于我们在学校里所学的十进制运算。惟一的区别是基数（radix）不是10，而是一个2的幂，例如2^{32}。

固定的和多精度算法在理论上稍有不同，并且大部分体现在实现上。在多精度算法中，我们通过分配所需的新内存来表示运算的结果，试图容纳任意大小的整数。这对于处理未知大小的数字很有用。但是，它有在计算过程中执行堆操作的开销，这可能会相当慢。固定精度算法只有一个有限的（固定的）空间来存储一个数的表示。正因为如此，它在计算过程中不需要堆操作。固定精度非常适合用于输入的大小是事先已知的任务中。例如，如果你准备实现ECC P-192，你知道最大的数为384位（2×192）并且你可以相应地进行调整。

在本文中，我们关注固定精度的实现细节，因为它们更加高效。我们会更多地讨论BigNum数学而不是让读者直接去看各种源代码。

更多的资源

对于想要了解更多内容的读者，可以参考各种其他的在更深层次上描述BigNum数学的资源。*BigNum Math*这本书通过介绍一个公开的数学算法库LibTomMath来讲述这一主题。对于刚接触这个方面知识的读者来说，很值得一读。*The Art of Computer Programming Volume* 2也从理论的角度讲述了这一主题，并讲解了高效的算法背后关键的数学概念。

读者也应该看一看可以免费获得的LibTomMath和TomsFastMath包的源代码，它们分别实现了多精度和固定精度运算。它们的源代码都有很好的说明文档，而且可以免费用于各种目的。

关键算法

当正确地实现了某些关键的算法时，一个BigNum算法库可以执行得相当好。即使一个问题会有多种算法，例如椭圆曲线的点加，但只有4种算法是最重要的：乘法（multiplication），平方（squaring）、约简（reduction）和模逆（modular inversion）。

当执行典型的公钥操作时，处理器时间的用量通常由最多到最少的顺序是约简、乘法、平方和模逆——这让你知道在优化的时候应该在哪里多花些时间。

实际上，我们所讨论的和教给小孩子们的算法是一样的，例如乘法。关键的区别是它们是怎样实现的。像累加器的结构、循环拆分和操作码的选择都可以造成这些不同。我们将使用适用于x86、PPC和ARM处理器的代码来讲解这4种算法，并使用GCC作为我们所选择的开发工具。

8.3 算法

8.3.1 表示

在研究高效的公钥密码所需要的算法之前，我们必须讨论用来表示这些整数的结构。针对我们的目的，我们将借助于TomsFastMath包，它使用了一种类似于下面的结构：

```
typedef struct {
   fp_digit dp[FP_SIZE];
```

```
    int        used,
               sign;
} fp_int;
```

其中*fp_digit*是一个表示单独数字的类型，一般等价于*unsigned long*，但可以根据平台来改变。和它一起的是我们还没有看到的*fp_word*类型。它是*fp_digit*的两倍大小并且能够存储两个最大的*fp_digit*值的乘积。*FP_SIZE*宏是默认的最大整数大小。它基于数字的大小以及所需的位数。例如，*fp_digit*是一个32位的类型，并且你希望表示384位的整数，那么*FP_SIZE*必须至少为12。

*used*标志表示数组*dp*中有多少个数字是非零的。这使得我们可以更高效地操作更短的整数。例如，虽然你的整数可以多达12个数字，但并不意味着它们都将会是那么大。对仅有6个数字的数执行12数字的操作将浪费时间和资源。

*sign*标志表示整数的符号。如果它为0，表示非负数；如果它是非零值，就表示负数。这意味着我们的整数总是无符号的而且只有这个标志控制这种状态。

*fp_int*类型的数字以little-endian的形式来使用。例如，*fp_digit*是32位的，数组*dp*包含{1, 2, 3, 4, 0, 0, 0, ..., 0}，那么这就表示整数$1+2\times2^{32}+3\times2^{64}+4\times2^{96}$。

8.3.2 乘法

乘法可以用多种算法来解决，这很像大多数的BigNum问题。像Karatsuba和Toom-Cook 线性乘法需要高效的渐近时间，但它们实际上并不是那样实用——至少对于典型的软件平台来说。它们在硬件平台上实用起来很方便。

可以证明乘以用于公钥算法中大小两个数的最好办法是基本$O(n^2)$时间复杂度的long-hand算法。即，如果我们用A乘以B，那么我们只要简单地把B的每个数字和A的每个数字相乘，并把所有的积累加（合计）起来。假设A和B具有相同的数字数（位数）*n*，这个处理需要n^2个单精度乘法运算（即*fp_digit*类型的乘法）（如图8-1所示）。

这个算法循环中的每次迭代生成乘积的一个数字（从第5步开始）。我们使用一种3个数字的累加器{*c*0, *c*1, *c*2}在内循环里累计乘积。先把2个数字的乘积加到累加器中，然后把进位放到第三个数字中。虽然听起来很复杂，但我们稍后会看到，它完全有效。

在内循环之外，累加器的最低数字保存未知的乘积数字。我们把*c*0保存到*C*中，然后把累加器下移——*c*1变成*c*0，*c*2变成*c*1，*c*2为0。

MIN()宏用来判断两个操作数中的最小值。在循环里使用它看起来像是冒着会产生分支的风险，而且在实际应用中，它确实如此。但是，正如我们将要看到的，循环拆分可以把它从代码中移除。这个实现的性能大多取决于有效执行内循环中的单一步骤的能力。

在我们来看让这切实可行的宏之前，先来研究一下TomsFastMath中的通用乘法函数。

Ripped from **fp_mul_comba.c:**

```
/* generic PxQ multiplier */
void fp_mul_comba(fp_int *A, fp_int *B, fp_int *C)
{
   int        ix, iy, iz, tx, ty, pa;
   fp_digit   c0, c1, c2, *tmpx, *tmpy;
   fp_int     tmp, *dst;

```

```
COMBA_START;
COMBA_CLEAR;
```

输入：

A, *B*: 整数

输出：

C: A和B的乘积

1. Zero *C*
2. $pa = A.used + B.used$
3. if $pa >= FP_SIZE$ then
    1. pa = FP_SIZE − 1
4. $\{c0, c1, c2\} = \{0, 0, 0\}$
5. for ix from 0 to $pa - 1$ do
    1. $ty = \text{MIN}(ix, B.used - 1)$
    2. $tx = ix - ty$
    3. $iy = \text{MIN}(A.used - tx, ty + 1)$
    4. for iz from 0 to $iy - 1$ do
        i. {c0, c1, c2} = {c0, c1, c2} + A.dp[tx + iz] * B.dp[ty − iz]
    5. C.dp[ix] = c0
    6. $\{c0, c1, c2\} = \{c1, c2, 0\}$
6. $C.sign = A.sign$ XOR $B.sign$
7. $C.used = pa$
8. Return *C*

图8-1 乘法算法

这两个宏启动我们的累加器并准备好算法的其余部分。在我们使用一些特殊寄存器（例如x86-sse2的XMM）的平台时，在执行这些代码之前，我们也许需要做一些特殊的事情（存储它们）。COMBA_START完全有可能是一个空的宏。

```
/* get size of output and trim */
pa = A->used + B->used;
if (pa >= FP_SIZE) {
   pa = FP_SIZE-1;
}

if (A == C || B == C) {
   fp_zero(&tmp);
   dst = &tmp;
} else {
   fp_zero(C);
   dst = C;
}
```

至此，这非常像我们的伪码（实际上伪码是衍生自这段代码）。

```
   for (ix = 0; ix < pa; ix++) {
      /* get offsets into the two bignums */
      ty = MIN(ix, B->used-1);
      tx = ix - ty;

      /* setup temp aliases */
      tmpx = A->dp + tx;
      tmpy = B->dp + ty;

      /* this is the number of times the loop will iterrate
         while (tx++ < a->used && ty-- >= 0) { ... }
       */
      iy = MIN(A->used-tx, ty+1);
```

此时，我们的内循环可以执行了。我们使用指针别名*tmpx*和*tmpy*来分别指向A和B的数字。这使得我们的内循环更为简单。

```
      /* execute loop */
      COMBA_FORWARD;
      for (iz = 0; iz < iy; ++iz) {
          MULADD(*tmpx++, *tmpy--);
      }
```

COMBA_FORWARD执行移动累加器的操作。MULADD执行{*c*0, *c*1, *c*2}+=**tmpx** *的运算。这时，我们还不知道这是怎样完成的。展示这种程序设计技术是为了说明宏的强大。

```
      /* store term */
      COMBA_STORE(dst->dp[ix]);
```

COMBA_STORE保存了累加器的最低数字。这是一个宏，因为累加器可以在一个特殊的GCC不能轻易看到的寄存器里。

```
   }
   COMBA_FINI;
```

COMBA_FINI是一个让我们进行清理工作的宏。例如，如果我们使用了MMX，那么我们可能不得不在这里放一个EMMS操作码。类似于COMBA_START，它也可以是一个空的宏。

```
   dst->used = pa;
   dst->sign = A->sign ^ B->sign;
   fp_clamp(dst);
   fp_copy(dst, C);
}
```

1. 乘法宏

现在，我们已经有了乘法的C语言代码，但是我们还不知道这个代码中最重要的部分是怎样实际工作的。我们灵巧地把它们的细节隐藏在一系列的C预处理宏里。为什么我们要这样做？为了让这个代码更难理解？为了让其他人更难使用？答案是多种多样的。

我们的宏机制是一个叫做TomsFastMath库的基础。它目前针对4种不同的体系，而不用重写代码的关键部分。我们的宏足够灵活，使得我们可以工作在32位和64位模式（也包括SSE2）的x86下，32位和64位的PPC以及ARMv4平台——即使x86、PPC和ARM的指令集并不相似。

首先，我们用这些宏的可移植的C实现来很好地了解它们所实现的功能。

```
#define COMBA_START
```

这个宏在乘法的逻辑中没有任何作用，只是让它能够更容易地支持各种平台。

```
#define COMBA_CLEAR \
   c0 = c1 = c2 = 0;
```

这个宏对累加器进行清零。

```
#define COMBA_FORWARD \

   do { c0 = c1; c1 = c2; c2 = 0; } while (0);
```

这个宏把累加器向右移并且在最高数字的位置插入一个0。

```
#define COMBA_STORE(x) \
   x = c0;
#define COMBA_STORE2(x) \
   x = c1;
```

这两个宏值保存在累加器之外。我们已经看到了COMBA_STROE，但还没有看到COMBA_STORE2，因为它是拆分代码的一部分。

```
#define COMBA_FINI
```

COMBA_FINID在这里只是为了支持而用，并且不是必需的。在本例中，它只是一个空的宏。

```
#define MULADD(i, j) \
   do { fp_word t; \
   t = (fp_word)c0 + ((fp_word)i) * ((fp_word)j); c0 = t; \
   t = (fp_word)c1 + (t >> DIGIT_BIT);c1 = t; c2 += t >> DIGIT_BIT; \
   } while (0);
```

这个宏执行了内循环中的乘法和累加。我们使用了一个双精度的*fp_word*来保存乘积并把它加到累加器中。通过使用双精度类型，我们避免了其他需要的溢出测试。这是因为C语言缺少一个“带进位加法”运算而需要传递进位。

至此，对于我们的代码所支持的所有平台，惟一需要修改的宏是MULADD。表8-1列出了各种平台和它们各自的宏。

表8-1 针对各种平台的MULADD宏

x86_32	`#define MULADD(i, j)\` `asm(\` `"movl %6,%%eax \n\t"\` `"mull %7 \n\t"\` `"addl %%eax,%0 \n\t"\` `"adcl %%edx,%1 \n\t"\` `"adcl $0,%2 \n\t"\` `:"=r"(c0), "=r"(c1), "=r"(c2)` `:"0"(c0),"1"(c1),"2"(c2),"m"(i),"m"(j)` `:"%eax","%edx","%cc");`
x86_64	`#define MULADD(i, j)\` `asm(\` `"movq %6,%%rax \n\t"\`

（续）

	``"mulq %7 \n\t"\```
	``"addq %%rax,%0 \n\t"\```
	``"adcq %%rdx,%1 \n\t"\```
	``"adcq $0,%2 \n\t"\```
	``:"=r"(c0), "=r"(c1), "=r"(c2)```
	``:"0"(c0),"1"(c1),"2"(c2),"m"(i),"m"(j)```
	``:"%rax","%rdx","%cc");```
x86_32 + SSE2	```#define MULADD(i, j)\```
	``asm(\```
	``"movd %6,%%mm0 \n\t"\```
	``"movd %7,%%mm1 \n\t"\```
	``"pmuludq %%mm1,%%mm0\n\t"\```
	``"movd %%mm0,%%eax \n\t"\```
	``"psrlq $32,%%mm0 \n\t"\```
	``"addl %%eax,%0 \n\t"\```
	``"movd %%mm0,%%eax \n\t"\```
	``"adcl %%eax,%1 \n\t"\```
	``"adcl $0,%2 \n\t"\```
	``:"=r"(c0), "=r"(c1), "=r"(c2)```
	``:"0"(c0),"1"(c1),"2"(c2),"m"(i),"m"(j)```
	``:"%eax","%cc");```
PPC32	```#define MULADD(i, j)\```
	``asm(\```
	``" mullw 16,%6,%7 \n\t" \```
	``" addc %0,%0,16 \n\t" \```
	``" mulhwu 16,%6,%7 \n\t" \```
	``" adde %1,%1,16 \n\t" \```
	``" addze %2,%2 \n\t" \```
	``:"=r"(c0), "=r"(c1), "=r"(c2)```
	``:"0"(c0),"1"(c1),"2"(c2),"r"(i),"r"(j)```
	``:"16");```
ARMv4	```#define MULADD(i, j)\```
	``asm(\```
	``"UMULL r0,r1,%6,%7 \n\t"\```
	``"ADDS %0,%0,r0 \n\t"\```
	``"ADCS %1,%1,r1 \n\t"\```
	``"ADC %2,%2,#0 \n\t"\```
	``:"=r"(c0), "=r"(c1), "=r"(c2)```
	``:"0"(c0),"1"(c1),"2"(c2),"r"(i),"r"(j)```
	``:"r0", "r1", "%cc");```

所有的5个宏实现同样的目标，但是具有不同的体系。它们都是用*i*乘以*j*并且把乘积累加在{*c*0, *c*1, *c*2}中。

x86_32和x86_64宏使用x86指令集中的MUL指令。它们把操作数*i*加载到EAX(RAX)寄存器中，然后执行一个*j*指向的*fp_digit*的乘法运算。x86指令集中的乘积总是存储在EDX：EAX（RDX：RAX）寄存器中。这个乘积然后被累加到3个数字的累加器{*c*0, *c*1, *c*2}中，我们让GCC把这个累加器作为处理器的寄存器。当GCC分析完这个宏，它会转化成类似于下面的汇编代码。

```
movq  (%rsp),%rax
mulq  8(%rsp)
addq  %rax,%rcx
adcq  %rdx,%rsi
adcq  $0,%rdi
```

对于那些不能读懂x86_64汇编的读者来说，我们可以告诉你GCC已经把{*c*0, *c*1, *c*2}换成{rcx, rsi, rdi}3个处理器寄存器了。这是让这段代码如此高效的部分原因。乘积的累计并没有额外的内存访问操作。

x86_32 SSE2代码是用于Pentium 4 Northwood（Prescott之前）处理器的。在这些处理器中，所有的整数MUL指令都是使用FPU，这意味着如果你简单地直接使用FPU，那么你就可以更快地得到乘积。Intel后来改变了他们的核心而且再也不是这种情况了。这段代码在ADM处理器上并不高效，至少比不上AMD的整数乘法器。

似乎使用SSE2并行地执行两个32位的乘法可能是更快的方法，但是，这并不是真的。AMD64（以及Opteron）系列处理器可以在大约5个处理器时钟周期内执行一个单独的64位乘法运算。如果用32位来完成的话，这将是64位的4倍。因此，即使你一次做两个操作，你也不得不在少于其一半时间内来完成以使得它变得更快。目前，SSE2乘法运算在ADM64处理器上还不是一个时钟周期，近期也不会是。

PPC32代码是另一个直接使用其指令集的。PPC和其他指令集的不同之处在于，每个操作码只产生乘积的一半。*mullw*指令产生乘积的低32位，而*mulhwu*产生高32位。我们可以把这两个乘法运算放在一起，但是，我们想避免破坏（clobbering）尽可能多的寄存器（clobbering是GCC的术语，它表示在汇编宏中将毁坏寄存器中的内容）。这个宏只破坏了一个寄存器，而且需要3个另外的寄存器用于累加器。

PPC64指令集具有类似的操作码。例如，*mulld*和*mulhdu*执行相应的64位乘法运算。这些目前还没有放到项目中，因为没办法接触到一个PPC64的机器。

ARMv4代码需要一个带有M特性的v4核心（例如ARM7TDMI）或者一个v5核心或者更高的。它可能是所有代码中最好的。我们把积存储在*r*0和*r*1中，然后累加。机敏的读者也许会注意到我们没有使用ARMv4“乘并累加”指令，这是因为它并不设置进位标志。因此，如果我们使用它的话，我们不能将每个数字存储32位。

2. 代码拆分

那么，现在我们已经拥有了通用的技术和宏来插入到代码中。我们需要为实际的运行性能对代码进行组织。真正的改进来自于拆分完整的内部和外部循环。如果我们事先知道所乘的数的大小，那么这就会有很大的好处。

现在，我们不用手动地拆分一个乘法运算，或者隐式地告诉GCC我们的数的大小，而是巧妙地构造一个程序，它是完全显式拆分的。为了做这件事，我们将写一个程序，它可以输出一个乘法函数的C源代码。我们将参考TomsFastMath项目中的另一个源文件。这个程序接受一个单一的规模N作为输入，并产生一个用于N × N乘法的C源代码。

```
ripped from comba_mult_gen.c:
011    /* program emits a NxN comba multiplier */
```

```
#include <stdio.h>

int main(int argc, char **argv)
{
   int N, x, y, z;
   N = atoi(argv[1]);

   /* print out preamble */
printf(
"void fp_mul_comba%d(fp_int *A, fp_int *B, fp_int *C)\n"
```

我们在知道它会处理多少个数字之后将对这个程序进行命名。例如，*fp_mul_comba16*()会执行一个16×16的乘法。

```
"{\n"
"   fp_digit c0, c1, c2, at[%d];\n"
"\n"
"   memcpy(at, A->dp, %d * sizeof(fp_digit));\n"
"   memcpy(at+%d, B->dp, %d * sizeof(fp_digit));\n"
```

我们把两个输入都复制到一个数组中，并不总是需要优化，但优化确实是降低在32位x86平台上所用寄存器数量的不错方法（因为第二个输入在*at*的第二半中的偏移是0）。

```
"   COMBA_START;\n"
"\n"
"   COMBA_CLEAR;\n", N, N+N, N, N, N);

   /* now do the rows */
   for (x = 0; x < (N+N-1); x++) {
printf(
"   /* %d */\n", x);
```

该外循环控制内循环对乘积的第*x*个数字的构造。

```
if (x > 0) {
printf(
"   COMBA_FORWARD;\n");
}
```

如果我们不是处于第一个乘积，那么就移动累加器，只有有一个完全拆分的循环优化才有可能实现。

```
      for (y = 0; y < N; y++) {
      for (z = 0; z < N; z++) {
          if ((y+z)==x) {
             printf("   MULADD(at[%d], at[%d]); ", y, z+N);
          }
      }
      }
```

这段代码以一种穷举的方法来构造内循环。幸运地是，我们只要执行它一次就能创建出源代码。实际上，我们遍历这两个输入，当找到它们的位置加起来等于*x*的时候，就执行一个MULADD操作。

```
printf(
"\n"
"   COMBA_STORE(C->dp[%d]);\n", x);
```

```
    }
printf(
"    COMBA_STORE2(C->dp[%d]);\n"
"    C->used = %d;\n"
"    C->sign = A->sign ^ B->sign;\n"
"    fp_clamp(C);\n"
"    COMBA_FINI;\n"
"}\n\n\n", N+N-1, N+N);

  return 0;
}
```

这段代码关闭这个函数，而且我们到此就结束了。这个程序的输出类似于下面的代码。

```
tom@box ~/libtom/tomsfastmath $ ./comba_mult_gen 2
void fp_mul_comba2(fp_int *A, fp_int *B, fp_int *C)
{
   fp_digit c0, c1, c2, at[4];
   memcpy(at, A->dp, 2 * sizeof(fp_digit));
   memcpy(at+2, B->dp, 2 * sizeof(fp_digit));
   COMBA_START;

   COMBA_CLEAR;
   /* 0 */
   MULADD(at[0], at[2]);
   COMBA_STORE(C->dp[0]);
   /* 1 */
   COMBA_FORWARD;
   MULADD(at[0], at[3]);    MULADD(at[1], at[2]);
   COMBA_STORE(C->dp[1]);
   /* 2 */
   COMBA_FORWARD;
   MULADD(at[1], at[3]);
   COMBA_STORE(C->dp[2]);
   COMBA_STORE2(C->dp[3]);
   C->used = 4;
   C->sign = A->sign ^ B->sign;
   fp_clamp(C);
   COMBA_FINI;
}
```

这个函数以一个完全拆分的形式执行一个2 × 2的乘法运算。拆分代码所得到的好处取决于数的大小以及待用的平台。在AMD64和Opteron处理器上，它总是很有益的，尤其是当内存充足的时候。例如，比较表8-2中Opteron的数据。

表8-2 AMD Opteron 整数乘法的时钟周期统计

大小（位）	拆分的（时钟周期）	循环的（时钟周期）
128	53	229
256	104	346
512	272	876
1024	934	2 248

从表8-2中我们可以看出，拆分的代码在Opteron上表现得非常好。实际上，它工作得如此之好以至于处理器在每个时钟周期内总共执行不止一个操作码，这是一个不错的性质。例如，

1 024位的乘法需要256个64位的乘法，这对它们自身而言应该需要1 280个时钟周期来处理。它具有更快的返回时间的原因是Opteron的乘法器是流行线形式的，这使得它一次可以处理不止1个的乘积。

在像PPC和ARM这样的平台上，内存通常都是不够用的，人们应该仔细的度量。虽然拆分的代码在性能上取得了成功，但由于它缺少控制结构（例如for循环），所以它常常很难存储代码。例如，一个6 × 6的乘法函数，它可能会用在ECC P-192中，在ARMv4处理器上如果完全拆分，需要1 088个字节（用GCC 4.1.1在一个ARM7TMDI平台上进行的测试）。大多数情况下这看起来不算多，但是这个函数的代码增长量是平方级的。例如，一个12 × 12的乘法函数，两倍的规模，却需要4倍的空间（4 036个字节）。

RSA算法（参见第9章）需要从32个数字（在32位的平台上）的范围开始。ARM上的一个32 × 32的乘法函数，当完全拆分时需要27 856个字节。这对于大多数的环境来说，是非常大的一个存储量。幸运地是，像ECC（参见第9章）这样的算法只使用很小的数，这使得用于乘法的循环拆分很容易处理。

作为一条好的通用规则，在大多数有效的RISC核心平台上，如果是处理10个或更少的数字，完全拆分循环是一种不错的折衷。当然，如果你想节省存储空间而且需要性能，它也几乎总是一种好的权衡。

8.3.3 平方

平方是乘法的一个特例，它经常出现在公钥运算中。不经意地看，平方只不过是把一个数乘以它自身。但是，存在一种非常简单的优化来执行这个运算。

假设我们正在乘以两个具有两个数字的数*ab*和*cd*。为了计算它们的乘积，我们需要求出乘积*ac*, *ad*, *bc*和*bd*。这看起来很简单。现在，如果*cd*等于*ab*又会怎样呢？那么，我们就是计算乘积*aa*, *ab*, *ba*和*bb*。从这里我们可以看出*ab*和*ba*值是相同的，只需要计算一次。总的来说，对于任意的*n*个数字的平方，需要计算（$n^2 + n$）/2个惟一的乘积。对于长度相等的整数来说，这使得平方大约比乘法快两倍。

类似于乘法算法，我们也使用一系列的可移植宏来让代码更加通用。平方比乘法稍微有点复杂，所以我们会使用更多的宏。首先，我们来看紧凑的通用平方代码（取自TomsFastMath）。

ripped from **fp_sqr_comba_generic.c:**

```
/* generic comba squarer */
void fp_sqr_comba(fp_int *A, fp_int *B)
{
  int       pa, ix, iz;
  fp_digit  c0, c1, c2;
  fp_int    tmp, *dst;
#ifdef TFM_ISO
  fp_word   tt;
#endif

  /* get size of output and trim */
  pa = A->used + A->used;
  if (pa >= FP_SIZE) {
     pa = FP_SIZE-1;
```

```
    }
```

这里我们在计算最终乘积的数字的数目。我们按照所要求的进行截短，以防止目的缓冲区溢出。

```
    /* number of output digits to produce */
    COMBA_START;
    CLEAR_CARRY;
```

像前面一样，我们用一个宏来初始化这个程序（COMBA_START），用另一个来清空累加器（CLEAR_COMBA）。我们使用和前面一样的3个寄存器作为累加器，不同的是这次我们在用法上作了一些调整。

```
    if (A == B) {
       fp_zero(&tmp);
       dst = &tmp;
    } else {
       fp_zero(B);
       dst = B;
    }

    for (ix = 0; ix < pa; ix++) {
        int      tx, ty, iy;
        fp_digit *tmpy, *tmpx;

        /* get offsets into the two bignums */
        ty = MIN(A->used-1, ix);
        tx = ix - ty;

        /* setup temp aliases */
        tmpx = A->dp + tx;
        tmpy = A->dp + ty;

        /* this is the number of times the loop will iterrate,
           while (tx++ < a->used && ty-- >= 0) { ... }
         */
        iy = MIN(A->used-tx, ty+1);

        /* now for squaring tx can never equal ty
         * we halve the distance since they approach
         * at a rate of 2x and we have to round because
         * odd cases need to be executed
         */
        iy = MIN(iy, (ty-tx+1)>>1);
```

这时，我们知道必须执行*iy*次才能产生平方的第*ix*个数字。

```
        /* forward carries */
        CARRY_FORWARD;
```

这个宏通过把3个寄存器的累加器向右移动一个寄存器来实现向前进位。

```
        /* execute loop */
        for (iz = 0; iz < iy; iz++) {
            SQRADD2(*tmpx++, *tmpy--);
        }
```

这是一个和我们前面看到过的MULADD相类似的循环，不同的是在这个例子中，*tmpx和

*tmpy的乘积加到累加器中两次。这个循环处理最终乘积中所有的重复项，通过执行一次乘法和对输出进行加倍来完成。

机敏的读者可能会注意到，我们可以保持各自独立的累加器并且在循环之外进行加倍操作。这稍后将出现。

```
        /* even columns have the square term in them */
        if ((ix&1) == 0) {
            SQRADD(A->dp[ix>>1], A->dp[ix>>1]);
        }
```

偶数数字输出在乘积中包含一项单一的平方。这实际上是前面的MULADD宏。

```
        /* store it */
        COMBA_STORE(dst->dp[ix]);
```

这个宏保存了累加器的最低数字。它并不移动累加器。

```
    }

    COMBA_FINI;
```

这个宏清除机器的状态（例如，如果你使用了MMX的话）。

```
    /* setup dest */
    dst->used = pa;
    fp_clamp (dst);
    if (dst != B) {
       fp_copy(dst, B);
    }
}
```

这个函数是通用的紧凑平方代码。如果编译时没有拆分循环，它可以非常紧凑，这在内存有限的嵌入式平台中很有用。但是，它不够快。简单地使用一个编译器开关来拆分并不会达到我们所寻求的速度。我们将再次写一个简单的用来生成平方函数的C源码生成器。

ripped from **comba_sqr_gen.c:**

```
#include <stdio.h>

int main(int argc, char **argv)
{
   int x, y, z, N, f;
   N = atoi(argv[1]);

printf(
"void fp_sqr_comba%d(fp_int *A, fp_int *B)\n"
"{\n"
"   fp_digit *a, b[%d], c0, c1, c2, sc0, sc1, sc2;\n"
"\n"
"   a = A->dp;\n"
"   COMBA_START; \n"
"\n"
"   /* clear carries */\n"
"   CLEAR_CARRY;\n"
"\n"
"   /* output 0 */\n"
"   SQRADD(a[0],a[0]);\n"
"   COMBA_STORE(b[0]);\n", N, N+N);
```

```

    for (x = 1; x < N+N-1; x++) {
printf(
"\n   /* output %d */\n"
"   CARRY_FORWARD;\n   ", x);

        for (f = y = 0; y < N; y++) {
            for (z = 0; z < N; z++) {
                if (z != y && z + y == x && y <= z) {
                   ++f;
                }
            }
        }

    if (f <= 2) {
        for (y = 0; y < N; y++) {
            for (z = 0; z < N; z++) {
                if (y<=z && (y+z)==x) {
                   if (y == z) {
                      printf("SQRADD(a[%d], a[%d]); ", y, y);
                   } else {
                      printf("SQRADD2(a[%d], a[%d]); ", y, z);
                   }
                }
            }
        }
    } else {
       // new method
       /* do evens first */
        f = 0;
        for (y = 0; y < N; y++) {
            for (z = 0; z < N; z++) {
                if (z != y && z + y == x && y <= z) {
                   if (f == 0) {
                      // first double
                      printf("SQRADDSC(a[%d], a[%d]); ", y, z);
                      f = 1;
                   } else {
                      printf("SQRADDAC(a[%d], a[%d]); ", y, z);
                   }
                }
            }
        }
        // forward the carry
        printf("SQRADDDB; ");
        if ((x&1) == 0) {
           // add the square
           printf("SQRADD(a[%d], a[%d]); ", x/2, x/2);
        }
      }
printf("\n   COMBA_STORE(b[%d]);\n", x);
    }
printf("   COMBA_STORE2(b[%d]);\n", N+N-1);

printf(
"   COMBA_FINI;\n"
"\n"
"   B->used = %d;\n"
```

```
090    "    B->sign = FP_ZPOS;\n"
091    "    memcpy(B->dp, b, %d * sizeof(fp_digit));\n"
092    "    fp_clamp(B);\n"
093    "}\n\n\n", N+N, N+N);
094
095      return 0;
096    }
```

这个程序生成用于各种输入大小的完全拆分的平方程序。我们可以看到一些和紧凑的平方中相同宏的名字。但是也有一些新的。

回忆一下前面我们提到的如何将累计的所有积加倍，已经证明我们可以在某些环境下对它进行优化。第38行的循环计算了对于给定的输出数字所需要的乘法的次数（这个计算方法是穷举；幸运地是，我们的输入是很小的，所以穷举对于给定的简单程度来说是一种不错的方法）。如果大于2的话，我们就把所有加倍的乘积累加起来，然后再对它们加倍一次。

为了完成这个操作，我们增加了几个新的宏。SQRADDSC执行一个乘法并把乘积放到一个辅助累加器中（同时把第三个寄存器清零）。SQRADDSC和MULADD做的事情一样，不同的是把乘积加到辅助累加器中。最后，SQRADDDB把辅助累加器加倍并把输出加到第一个累加器中。

我们来检验一个8个数字平方的输出。

```
void fp_sqr_comba8(fp_int *A, fp_int *B)
{
   fp_digit *a, b[16], c0, c1, c2, sc0, sc1, sc2;
   a = A->dp;
   COMBA_START;

   /* clear carries */
   CLEAR_CARRY;

   /* output 0 */
   SQRADD(a[0],a[0]);
   COMBA_STORE(b[0]);

   /* output 1 */
   CARRY_FORWARD;
   SQRADD2(a[0], a[1]);
   COMBA_STORE(b[1]);
<snip>
   /* output 5 */
   CARRY_FORWARD;
   SQRADDSC(a[0], a[5]); SQRADDAC(a[1], a[4]); SQRADDAC(a[2], a[3]); SQRADDDB;
   COMBA_STORE(b[5]);
<snip>
```

从代码中我们可以看出，这个程序识别出第5个输出需要3个乘积并使用了辅助寄存器。你也许会感到奇怪，为什么我们只对3个或更多的项执行这种优化。这是因为我们总是至少执行两次加法——一次是把辅助累加器加倍，一次是把它加到第一个累加器上。回忆一下，我们并不只是简单地把第一个累加器加倍，因为它有平方项（偶数列，使用SQRADD），它们不能被加倍。

我们来比较一下紧凑的和拆分的代码，如表8-3所示。

使用代码拆分再次节省了大量的时钟周期。代码大小的增长也是平方级的。在本例中，代码的增长为$n^2/2$，即如果输入大小增长了n-fold，那么代码大小大概增长$n^2/2$-fold（如表8-4所示）。

表8-4表明平方代码大小的增长在实际应用中仍然接近于平方级数。

表8-3 AMD Opteron 整数平方的时钟周期统计

大小（位数）	拆分的	循环的
128	40	223
256	78	338
512	231	722
1024	652	2 145

表8-4 64位的x86上的拆分平方代码大小

数字的数目	代码大小（字节）
4	458
8	1 169
16	3 490
32	13 138

平方宏

我们用于平方的宏是简单地从乘法宏衍生而来的。为了节省版面，读者可以参照免费的TomsFastMath包。在文件fp_sqr_comba.c中，读者会发现用于所有支持的平台的等价宏。但是，为了完整起见，我们来看一看这些宏的ISO C的实现。

```
#define CLEAR_CARRY \
   c0 = c1 = c2 = 0;
```

这是把3个累加寄存器清零（在本例中，它们是位于栈中的变量）。

```
#define COMBA_STORE(x) \
   x = c0;
```

这是保存累加器的最低寄存器。

```
#define COMBA_STORE2(x) \
   x = c1;
```

这是保存累加器的中间一个寄存器。

```
#define CARRY_FORWARD \
   do { c0 = c1; c1 = c2; c2 = 0; } while (0);
```

把累加器向右移动一个寄存器，在最高寄存器的位置插入一个零。注意do-while结构的使用，它可以让这个宏安全地插到任何地方。

```
/* multiplies point i and j, updates carry "c1" and digit c2 */
#define SQRADD(i, j)                                  \
   do { fp_word t;                                     \
   t = c0 + ((fp_word)i) * ((fp_word)j);  c0 = t;     \
   t = c1 + (t >> DIGIT_BIT);             c1 = t; c2 += t >> DIGIT_BIT; \
   } while (0);
```

这是把i和j相乘并把结果加到累加器一次。

```
/* for squaring some of the terms are doubled... */
#define SQRADD2(i, j)                                 \
   do { fp_word t;                                     \
   t  = ((fp_word)i) * ((fp_word)j);                   \
   tt = (fp_word)c0 + t;                   c0 = tt;    \
   tt = (fp_word)c1 + (tt >> DIGIT_BIT); c1 = tt;    \
```

```
c2 += tt >> DIGIT_BIT;                                   \
tt = (fp_word)c0 + t;                   c0 = tt;         \
tt = (fp_word)c1 + (tt >> DIGIT_BIT); c1 = tt;           \
c2 += tt >> DIGIT_BIT;                                   \
} while (0);
```

这执行了和SQRADD同样的运算，但不是把输出加一次，而是加两次。我们并不把乘积加倍，因为它不适用于一个fp_word变量。我们将被强制加两次。

```
#define SQRADDSC(i, j)                             \
   do { fp_word t;                                  \
      t =  ((fp_word)i) * ((fp_word)j);             \
      sc0 = (fp_digit)t;                            \
      sc1 = (t >> DIGIT_BIT); sc2 = 0;              \
   } while (0);
```

这个宏执行的是乘上输入并把它存储到一个新的辅助累加器中。我们使用变量{*sc*0, *sc*1, *sc*2}而不是默认的，因为我们想在加倍之前把这些乘积单独累加。

```
#define SQRADDAC(i, j)                          \
   do { fp_word t;                               \
   t = sc0 + ((fp_word)i) * ((fp_word)j);\
   sc0 = t;                                      \
   t = sc1 + (t >> DIGIT_BIT);                   \
   sc1 = t;                                      \
   sc2 += t >> DIGIT_BIT;                        \
   } while (0);
```

这个宏执行SQRADD宏，但它把乘积加到辅助累加器中。

```
#define SQRADDDB                                                        \
   do { fp_word t;                                                       \
   t = ((fp_word)sc0) + ((fp_word)sc0) + c0;                             \
   c0 = t;                                                               \
   t = ((fp_word)sc1) + ((fp_word)sc1) + c1 + (t >> DIGIT_BIT);          \
   c1 = t;                                                               \
   c2 = c2 + ((fp_word)sc2) + ((fp_word)sc2) + (t >> DIGIT_BIT); \
   } while (0);
```

最后，这个宏把辅助累加器加倍并把它加到第一个中。这些辅助宏使得我们可以累加这些加倍的乘积，而不用冗繁地对它们加倍。

8.3.4 Montgomery约简

对于一个有能力的BigNum库所需要的最后一个性能关键算法是Montgomery约简。模约简是指计算一个除法的余数的过程，特别地，我们是求一个整数除以另一个叫做模数（modulus）的余数。

在公钥算法中，我们将遇到一些特殊的环境。我们用来进行相除的数通常不会大于模的平方。从这个事实中，我们可以构造出各种优化的约简算法。在本例中，我们将研究Montgomery约简，因为它是高效且通用的。如下的代码取自TomsFastMath并执行了通用紧凑的Montgomery约简。

```
/* computes x/R == x (mod N) via Montgomery Reduction */
```

```
void fp_montgomery_reduce(fp_int *a, fp_int *m, fp_digit mp)
{
   fp_digit c[FP_SIZE], *_c, *tmpm, mu;
   int      oldused, x, y, pa;

   /* bail if too large */
   if (m->used > (FP_SIZE/2)) {
      return;
   }
```

由于我们使用的是固定精度，所以我们把输入模数长度限制在BigNum长度的一半。

```
   pa = m->used;

   /* copy the input */
   oldused = a->used;
   for (x = 0; x < oldused; x++) {
       c[x] = a->dp[x];
   }
   for (; x < 2*pa+1; x++) {
       c[x] = 0;
   }
```

此时，我们已经复制了要找出其余数的输入的低半部分。这等价于把输入模2的幂约简，这是Montgomery约简的第一步。

```
   MONT_START;
```

这个宏允许我们能够做内循环需要的任何初始化工作。

```
   for (x = 0; x < pa; x++) {
       fp_digit cy = 0;
       /* get Mu for this round */
       LOOP_START;
```

LOOP_START宏用*c*[*x*]乘以*mp*来创建循环的这次迭代的Mu个数字。它可以用汇编代码来实现，但正如我们将要看到的，在大多数平台上只用C也是很容易实现的。

```
       _c   = c + x;
       tmpm = m->dp;
       y = 0;

       for (; y < pa; y++) {
          INNERMUL;
          ++_c;
       }
```

内循环执行一个简单的单数字乘法和累加。我们可以在x86_64平台上通过推迟内存加载操作来对它进行一些拆分，但是，现在我们还是让它简单一些。

```
       LOOP_END;
```

LOOP_END宏只用来取*cy*寄存器。即，如果LOOP_START和INNERMUL把*cy*关联到一个C编译器看不到的机器寄存器，那么这个宏就会取出它。在这个语句之后，变量*cy*必须和机器寄存器的内容同步（如果有的话）。

```
       while (cy) {
           PROPCARRY;
```

```
            ++_c;
        }
```

这个循环把前位向前传递。PROPCARRY宏执行了把*cy*的进位传递到_*c*[0]中的任务。_*c*指针是递增的，而且只要*cy*非零，那么循环就会重新迭代。

```
    }

    /* now copy out */
    _c    = c + pa;
    tmpm = a->dp;
    for (x = 0; x < pa+1; x++) {
        *tmpm++ = *_c++;
    }

    for (; x < oldused; x++)    {
        *tmpm++ = 0;
    }
```

此时，我们已经复制出来了余数并把高的数清零。

```
    MONT_FINI;
```

这个宏是用来按照要求对机器状态进行清理的。

```
    a->used = pa+1;
    fp_clamp(a);

    /* if A >= m then A = A - m */
    if (fp_cmp_mag (a, m) != FP_LT) {
      s_fp_sub (a, m, a);
    }
}
```

最后的一点代码对BigNum进行压缩以删除不使用的数字，并且减去模数以避免在Montgomery约简中可能出现的“off by one（偏移一个单位）”情况。

1. Montgomery 约简拆分

可以证明这段代码加上最理想的平台上的宏，即使在拆分代码的情况下也不能产生很好的效果。也就是说，性能上的收获几乎不能抵消代码的增长。但是，如果你非要提升性能的话，可以研究TomsFastMath中的源码，它使用拆分了的Montgomery约简来去除最后2%的性能。

2. Montgomery宏

表8-5中列出了用于x86、ARM和PPC的Montgomery约简宏。它们都使用了一种基于C的LOOP_START宏，定义如下。

```
#define LOOP_START \
   mu = c[x] * mp
```

表8-5　用于x86的Montgomery 约简宏

INNERMUL	`#define INNERMUL \`
	`asm( \`
	`"movl %5,%%eax \n\t" \`
	`"mull %4 \n\t" \`
	`"addl %1,%%eax \n\t" \`

（续）

```
                  "adcl $0,%%edx \n\t"              \
                  "addl %%eax,%0 \n\t"              \
                  "adcl $0,%%edx \n\t"              \
                  "movl %%edx,%1 \n\t"              \
              :"=g"(_c[LO]), "=r"(cy)               \
              :"0"(_c[LO]), "1"(cy), "g"(mu), "g"(*tmpm++) \
              :"%eax", "%edx", "%cc")
PROPCARRY     #define PROPCARRY                     \
              asm(                                  \
                  "addl    %1,%0      \n\t"         \
                  "setb    %%al       \n\t"         \
                  "movzbl %%al,%1    \n\t"          \
              :"=g"(_c[LO]), "=r"(cy)               \
              :"0"(_c[LO]), "1"(cy)                 \
              :"%eax", "%cc")
```

目前惟一的不同就是x86_32的SSE2代码，它使用了一个SSE2乘法而不是标准的整数ALU乘法。

我们的x86代码用setb和movzbl指令来执行一个无分支的把进位存储到*cy*中的操作。如果进位标志设置了，第一条指令setb就把*al*寄存器设为1，否则为0。第二条指令movzbl会把字节寄存器*al*零扩展到保存变量*cy*的寄存器中。这些指令是非常有用的，因为它们使得我们可以避免分支，这会泄露侧信道信息并妨碍性能（如表8-6所示）。

表8-6 用于ARMv4的Montgomery约简宏

```
INNERMUL      #define INNERMUL                      \
              asm(                                  \
                 " LDR     r0,%1          \n\t" \
                 " ADDS    r0,r0,%0       \n\t" \
                 " MOVCS   %0,#1          \n\t" \
                 " MOVCC   %0,#0          \n\t" \
                 " UMLAL   r0,%0,%3,%4    \n\t" \
                 " STR     r0,%1          \n\t" \
              :"=r"(cy),"=m"(_c[0])
              :"0"(cy),"r"(mu),"r"(*tmpm++),"1"(_c[0])
              :"r0","%cc");
PROPCARRY     #define PROPCARRY                     \
              asm(                                  \
                 " LDR    r0,%1          \n\t" \
                 " ADDS   r0,r0,%0       \n\t" \
                 " STR    r0,%1          \n\t" \
                 " MOVCS  %0,#1          \n\t" \
                 " MOVCC  %0,#0          \n\t" \
              :"=r"(cy),"=m"(_c[0])
              :"0"(cy),"1"(_c[0])
              :"r0","%cc");
```

类似于x86的情况，我们使用条件移动指令MOVCS和MOVCC来避免代码中的分支。这对指令的工作很类似于x86指令中的CMOV类。当进位标志已设置时，MOVCS会执行移动，而

MOVCC是在当进位没有设置的时候才执行移动。虽然分支在ARM处理器上算不上是一个危险，但它仍然会泄露侧信道数据而且应该被避免。

作为一种性能上的考虑，可以使用UMLAL指令来执行一个32 × 32=>64的乘法和累加运算。它使得我们可以把3条指令组合到一条中并且节省许多时钟周期。但我们不能把它用在乘法和平方代码中，因为UMLAL不会设置进位标志——这相当不幸，因为它可以提高ARM BigNum数学运算的速度（如表8-7所示）。

表8-7　用于PPC的Montgomery 约简宏

INNERMUL

```
#define INNERMUL                              \
asm(                                          \
   " mullw     16,%3,%4          \n\t"   \
   " mulhwu    17,%3,%4          \n\t"   \
   " addc      16,16,%0          \n\t"   \
   " addze     17,17             \n\t"   \
   " lwz       18,%1             \n\t"   \
   " addc      16,16,18          \n\t"   \
   " addze     %0,17             \n\t"   \
   " stw       16,%1             \n\t"   \
:"=r"(cy),"=m"(_c[0])
:"0"(cy),"r"(mu),"r"(tmpm[0]),"1"(_c[0])
:"16", "17", "18","%cc"); ++tmpm;
```

PROPCARRY

```
#define PROPCARRY                             \
asm(                                          \
   " lwz       16,%1             \n\t"   \
   " addc      16,16,%0          \n\t"   \
   " stw       16,%1             \n\t"   \
   " xor       %0,%0,%0          \n\t"   \
   " addze     %0,%0             \n\t"   \
:"=r"(cy),"=m"(_c[0])
:"0"(cy),"1"(_c[0])
:"16","%cc");
```

8.4 总结

8.4.1 核心算法

我们只研究了3种在执行公钥运算时占用大量处理器时间的核心算法。它们不能形成执行公钥运算所需要的所有函数的一个详尽的列表。

除了基本的加法、减法和比较，还需要乘法逆、模幂和最大公因子运算。这些算法并不特定于某些平台，而且在其他几本书中也做了详细的介绍。

有几种求乘法逆的方法，例如基于扩展欧几里德算法（extended Euclidean algorithm）和殆逆元（almost inverse）的算法。欧几里德算法是目前最通用和常见的算法，但它并不是最快的。殆逆元算法对于模数为奇数（尤其是最低位）以及硬件实现的情况是很理想的。通常认为是Kaliski在“Almost Montgomery Inversion”中提出的这一算法，但它也是一篇1995年的密码学论文的一部分，其中他们提出了一种可以快速计算一个大约等于真正的模逆值的算法

(Richard Schroeppel, Hilarie Orman, Sean O' Malley, Oliver Spatscheck, "Fast KeyExchange with Elliptic Curve Systems", 1995, Advances in Cryptology-Crypto' 95, Edited by Don Coppersmith, Springer-Verlag)。最终的逆元是由近似值以有效地移位和加法运算来计算的。

模幂是另外一种可以根据条件用各种算法来解决的问题。*The Handbook of Applied Cryptography*（中文译作《应用密码学手册》）略述了几种，例如基本的left-to-right幂算法、窗口幂算法以及向量链接幂算法。这些算法在Knuth所写的*The Art of Computer Programming Volume* 2（中文译为《计算机程序设计的艺术第2卷》中以更为理论的形式讲解。他的书讨论了各种幂算法的渐近行为，该知识是合理地开发了一个可以广泛应用的数学库的基础。更为实用的针对幂的讨论在Tom St Denis所写的书*BigNum Math: Implementing Cryptography Multiple Precision Arithmetic*中可以找到。这本书包含了许多对实现独立的BigNum库有用的源代码。

8.4.2 大小与速度

除了采取适合于平台的算法之外，人们经常会提到为了更好地使用代码和数据内存该做出怎样的选择。正如我们所看到的，循环拆分可以在很大程度上加速整数乘法和平方。但是，速度的增加是以大小的牺牲为代价的。我们用于乘法和平方的算法是平方级数的（例如$O(n^2)$）而且其收益也是如此（从技术上说，像Karatsuba和Toom-Cook乘法这样的算法并不是平方级数的。但是，它们对于我们所要使用的数的规模完全没有效果）。

各种算法，例如那些用于加法、减法和移位的算法，都是线性的（即$O(n)$），而且可以用代码拆分来相对比较经济地提高速度。从整体上来看，在执行公钥运算时这是可以节省时钟周期的。但是，在大部分算法中这种节约（类似于代价）并不明显。使用像殆逆元这样的算法并且进行拆分确实是合算的，它们不使用乘法而且执行一个$O(n^2)$数量的移位和加法运算。一个好的用来表示拆分线性运算是不是一个好主意的标志是，计算求模逆运算所花费的时间。如果比较起来它是有效的，那么显然拆分它们是一个好主意。

一种通常的好的做法是，如果你的数中有10个或者更少的数字（例如在一个32位的平台上的320位的数），强烈建议考虑循环拆分作为一种性能上的权衡。通常此时，乘法所需要的设置代码（即在内循环之前或之后的代码）消耗的是实际由执行乘法所花的时钟周期。保持小的乘法函数是紧凑的会节省存储空间，但处理的效率下降是很大的。当然，这仍然取决于可用的内存，但经验表明它通常是一个好的起点。10个数字以上和拆分的代码通常在嵌入式平台上大得很难管理，而且你必须花费更多的时间用于内循环中。

大致来说，如果我们令c表示管理内循环所花费的时钟周期，n表示数字的数目，让循环紧凑所造成的性能损失为nc/n^2，或者简化为c/n。这还没有考虑实际执行循环（例如分支，递减计数器等）所造成的性能损失。

8.4.3 BigNum库的性能

幸运地是，对于大多数的开发者而言，有一些可用的高效的数学库。已经有了一两个可以直接采用的库，因为强烈建议开发者不要写他们自己的数学库，除非是为了学术上的练习。

写一个完整并且功能很强的数学库需要经验也需要技巧，因为每个程序都需要测试许多极

端情况。随着性能要求的提高，代码通常没有要求的那样直接，尤其是当在等式中引入汇编代码时。

1. GNU 多精度算法库

GNU多精度（GNU Multiple Precision, GMP）算法库是目前为止最悠久并且最知名的用于处理任意长度整数、有理数和浮点数的算法库。这个算法库以GPLv2许可的方式发布的，官方主页为www.swox.com/gmp/。

这个算法库为了各种任务而设计，其中很少有函数本身是为密码学而设计的。GMP的目标是高效地，按照渐近的顺序，处理尽可能广的各种输入大小。有一些算法只有当输入大到接近万位数的长度时才有用。

即使拥有方便的灵活性，但这个算法库仍然对密码学所用大小的数很有效。它有很好的优化了的乘法、平方和幂运算代码，这使得它具有很高的竞争力。它也适用于各种机器，这使得它是平台独立的。

GMP最明显的失败之处是它的大小。这个库有几MB的大小而且很难再调整。另一个明显的失误是缺少一个公用的可以访问的Montgomery约简函数。这使得用它来实现椭圆曲线密码学更为困难，因为人们不得不写一个自己的约简函数来执行点运算。

2. LibTomMath 算法库

LibTomMath是一个构建相当好的算法库。它使用教学的思想进行设计并且使用可移植的ISO C语法进行编写。虽然它并不是世界上最快的数学算法库，但它是非常独立于平台并且紧凑的。根据数的规模和待定的操作，它可以达到像GMP和TomsFastMath这样的算法库30%～50%的性能。

LibTomMath包的官方主页是http://math.libtomcrypt.com而且是公开的。也就是说，它可以免费用于各种目的，而且使用这个库不需要任何授权许可。由于这个库遵循ISO C标准，所以它形成了各种可移植的项目中的集成部分，包括，Tcl脚本语言。

LibTomMath比GMP的函数要少，但它有足够的函数用来构造大多数的公钥算法。这个代码使用类似于GMP的多精度表示，这使得它可以通过容纳在编译时并不知道规模的数来解决各种任务。

8.4.4 TomsFastMath算法库

TomsFastMath是由LibTomMath的作者所设计的一个较新的数学库。它具有非常相似的API，但代码是压缩并且优化了的，主要是用于快速密码数学。这个库使用可观的代码拆分，这使得它即快又大。幸运地是，它在构建时可以通过配置来适应在内存有限平台上的给定问题（例如，RSA-1024或者ECC P-192）。

TomsFastMath包的主页是http://tfm.libtomcrypt.com，而且是公开的。它针对32位和64位的x86平台、PPC平台以及ARMv4以更高的处理器进行了优化。这个包可以用ISO C模式进行构建，但在这种模式下并不快。这个项目并不像LibTomMath那样争取到很多的可移植性，但用原速度来弥补了这种不足。

不像LibTomMath和GMP，TomsFastMath并不是一个通用目的的库。它的设计主要是用于密

码学的任务，而且做了各种设计上的决定，例如使用固定精度的表示。这意味着，你必须在编译这个库之前事先知道你要用的最大的数。此外，模逆和求逆的程序只接受奇数的模。这是因为偶数模并不用于任何标准的公钥算法中，而且像这种情况在这个项目中不值得花时间来考虑。

8.5 常见问题

下面的常见问题，由本书的作者所回答，它们即可以用来测试你对本章所出现的概念的理解，也可以帮助你在现实生活中实现这些概念。如果希望作者解答你的问题，请浏览www.syngress.com/solutions，然后点击“Ask the Author”表单。

问：什么是BigNum数学？

答：多精度或者固定精度数学是允许表示的操作大整数的算法集，通常设计用来补偿缺少对大整数的内在支持的情况。这些算法使用更小的、一般是固定的整数（常叫做零件或者数字）来表示大整数。

问：为什么了解BigNum数学是重要的？

答：大整数形成了像RSA、ElGamal和椭圆曲线密码学等公钥算法的基础。这些算法需要大数来使得像分解和离散对数这样的攻击无效。例如，RSA需要至少在1 024位范围内的数，而ECC需要至少192位的数。这些值不可能被像C、C++和Java等语言的内置变量所支持。

问：哪些算法是最值得优化的？

答：这个问题的答案取决于你所使用的公钥算法的类型。在大部分情况下，你会需要快速的Montgomery约简、乘法和平方。优化的不同之处在于数的规模。像ECC这样的算法可以从小的拆分的算法中获得好处，而像RSA和ElGamal这样的算法当内存足够时可以从大的拆分的算法中获得好处。对于ECC，我们想使用快速的定点算法，而对于RSA，我们会使用滑动窗口模幂算法（参见第9章）。

问：哪些库提供了公钥算法所需要的算法？

答：GNU MP（GMP）提供了针对很大范围的输入规模的各种数学算法。它以GPL许可提供，网站是www.swox.com/gmp/。LibTomMath针对可变范围的输入规模而提供了多种密码学相关的算法。它并不像GMP那样通用，主要是针对密码学的任务而设计的。它是公开的，可以在网站http://math.libtomcrypt.com获得。TomsFastMath主要是针对速度来设计的，它提供了一个更为有限的密码学相关算法的子集。它比LibTomMath快许多而且在速度方面经常等于或者好于GMP。它是公开的，可以在网站http://tfm.libtomcrypt.com获得。

第 9 章

公钥算法

本章解决方案：

- 什么是公钥密码
- 公钥密码的目标
- 标准RSA密码
- 标准椭圆曲线密码学
- 公钥算法
- 进一步的参考
- ☑ 总结
- ☑ 快速查找解决方案
- ☑ 常见问题

9.1 简介

到目前为止，我们已经讨论了对称密钥算法，例如AES、HMAC、CMAC、GCM和CCM。这些算法叫做对称（或共享秘密）算法，因为所有的成员都共享同样的密钥值。暴露这个密钥将会危及系统的安全。这意味着我们已经假设我们共享一个密钥，现在我们将回答怎样共享这一问题。

公钥算法，也叫作非对称密钥（asymmetric key）算法，用来（主要）解决两个对称密钥算法不能解决的问题：密钥分发（key distribution）和不可否认（nonrepudiation）。第一个可以帮助解决保密性的问题，后者可以帮助解决真实性的问题。

公钥算法通过非对称的操作来完成这些目标；也就是说，一个密钥被分成两个相应的部分，一个公钥和一个私钥。公钥之所以这么命名是因为它可以安全地公开分发给所有需要它的人。公钥使得人们可以加密消息以及验证签名。私钥之所以这么命名是因为它必须保持私有而且不能分发。私钥通常只被一个人所有或者被大多数环境下的一台设备所有，但是在技术上可以在一个可信任的团体中共享。私钥可以解密消息并生成签名。

第一个公开披露的公钥算法是Diffie-Hellman密钥交换算法，至少一开始它允许已知成员之间的密钥分配。ElGamal把它扩展为一个完整的加密和签名公钥体系，并用于ECC加密，我们稍后将会看到。在Diffie-Hellman公布不久之后，另一个叫作RSA（Rivest Shamir Adleman）的算法公开发表。RSA既能够加密也能够签名，而只需使用ElGamal一半的带宽。后来，RSA以各种形式成为标准。

随后，在20世纪80年代，椭圆曲线被提议作为一个ElGamal加密和DSA可以在上面执行的阿贝尔群（abelian group），并且在20世纪90年代到21世纪初之间，各种算法被提出，它们使得

椭圆曲线密码学（elliptic curve cryptography）成为一个吸引人的代替RSA和ElGamal的算法。

针对本书的目的，我们将讨论PKCS #1标准RSA和ANSI标准ECC密码学。它们是NIST为公钥密码所指定的3个标准算法中的两个，而且一般也代表了商业部门的需求。

9.2 公钥密码的目标

公钥密码用来解决各种对称密钥算法不能解决的问题。特别地，它可以用来提供保密性和不可否认。保密性通常由密钥分发和一个对称密钥密码算法来提供。这被称为混合加密（hybrid encryption）。不可否认常由数字签名和一个散列函数来提供。

9.2.1 保密性

保密性是使用公钥算法以两种形式中的任一种来实现的。第一种方法是只使用公钥算法把明文编码为密文（如图9-1所示）。例如，RSA可以接受一个短的明文并直接对它加密。如果应用程序必须只加密短的消息，这就是有用的。但是，这种方便是以牺牲速度为代价的。正如我们稍后将看到的，公钥操作比它们相应的对称密钥算法要慢得多。



图9-1 公钥加密

实现保密性的第二种有用的方法是用一种叫做混合加密的模式。这种模式结合了公钥加密的密钥分发优势以及对称算法的速度优势。在这种模式中，每个加密了的消息通过如下的步骤来处理，首先选择一个随机对称密钥，用公钥算法对它进行加密，最后用随机的对称密钥对消息进行加密。密文就是随机公钥和随机对称密钥密文的组合。

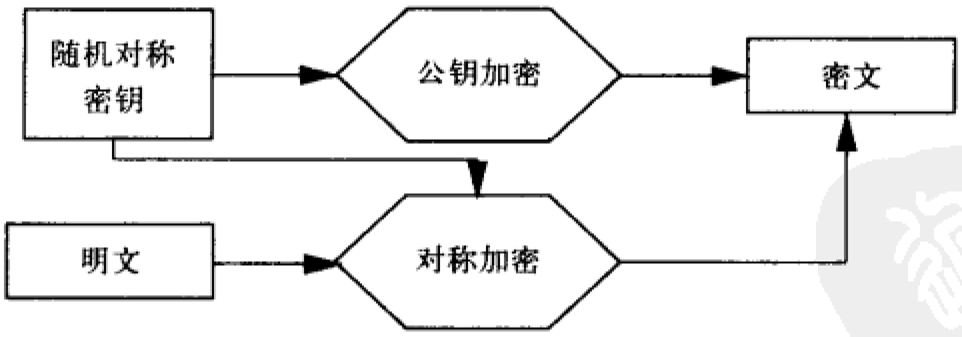


图9-2 混合加密

9.2.2 不可否认和真实性

不可否认是不能否认或拒绝承担一个协议的义务的性质。在以纸张为主的世界里，这是通过在合同上用手签名来完成的。它们有在实际中很难被伪造的性质，至少对于一个外行是这样的。在数字世界里，它们是由使用一个私钥的公钥签名来产生的。相应的公钥可以验证签名。通过从一个信任的成员来验证一个签名，也可以使用签名来测试消息的真实性。

在通常以X.509证书（SSL、TLS）和PGP密钥的形式使用的公钥基础设施（Public Key

Infrastructure，PKI）中，一个公钥可以由一个所有用户共有的作为担保实体的机构（authority）来签名。例如，在SSL和TLS领域中，VeriSign就是许多根认证机构（root certificate authority）之一。使用SSL的应用程序一般都在本地安装来自VeriSign的公钥，这样它们就可验证其他VerSign所担保的公钥。

不管签名的目的是什么，它们都是用一种通用的形式来生成的。被认证的消息首先用一个安全的密码学散列函数进行散列，然后把消息摘要发送给公钥签名算法（如图9-3所示）。

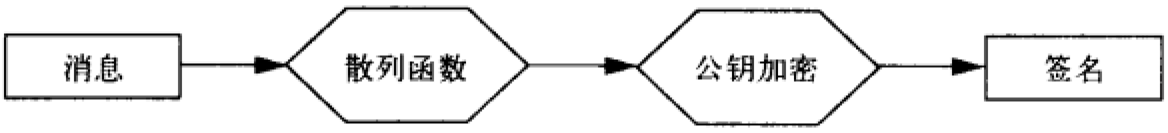


图9-3 公钥签名

这种公钥签名结构要求散列函数是抗碰撞的。如果它很容易就能找到碰撞，则可以简单地通过产生具有相同散列的文档来伪造签名。由于这个原因，我们通常把散列大小和强度与底层公钥算法的强度相符合。例如，使用一个只需要2^{64}的运算就可以攻破SHA-256的公钥算法是没有什么用的。攻击者不用找到碰撞就可攻破公钥并产生签名。

9.3 RSA公钥密码

RSA公钥密码是基于取第e个根模一个合数的问题。如果你不知道这是什么意思，没关系，稍后将会进行解释。RSA在众多公钥算法中是惟一一个直接把消息（或者是签名中的消息摘要）变换为公钥原型的算法。与求这个变换的逆的困难性一样（这被称为陷门（trapdoor）），它不能直接用一种安全的方式来使用。

为了解决这个问题，RSA Security（公司名）制定了一种标准，叫做公钥密码标准（Public Key Cryptographic Standards，PKCS），他们早期的第一个标准#1详细地描述了怎样使用RSA。这个标准包括了怎样填充数据，使得你可以用一种安全的方式来使用RSA变换。这个标准有两个流行的版本：v1.5和当前的v2.1。旧的版本技术上在大多数环境下还是安全的，但没有较新的版本好，它解决了v1.5中不安全的几种情况。由于这种原因，不建议在新的系统中实现v1.5。令人遗憾地是，情况并非如此。SSL仍然使用v1.5的填充，用来处理多种基于PKI的应用程序。

RSA曾经是一个美国的专利，但现在已经过期了。最初的RSA，包括PKCS #1填充，现在完全是专利免费的。虽然各种RSA的变形，例如多素数（multiprime）RSA仍然是专利的，但这些并不常用而且常为各种安全问题而被避免使用。

9.3.1 RSA简述

我们现在简要地讨论一下RSA所用的数学背景知识，这样本章的其余部分才有合理的意义。

1. 密钥生成

RSA的密钥生成从概念上来看是一个简单的过程，但它在实际中并不简单，尤其是对于寻求速度的实现（如图9-4所示）。

输入:

e: 公开的指数

n: 需要的模数的位长

输出:

n: 公开的模数

d: 私有的指数

1. 选择一个随机的素数p，长度为$n/2$位，使得$gcd(p\text{-}1, e) = 1$
2. 选择一个随机的素数q，长度为$n/2$位，使得$gcd(q\text{-}1, e) = 1$
3. 令$n = pq$
4. 计算$d = e^{-1} \bmod (p\text{-}1)(q\text{-}1)$
5. 返回n, d

图9-4 RSA密钥生成

指数e必须是奇数，因为$p-1$和$q-1$都有2这个因子。典型的e为3、17或者65 537，它们都可以有效地用作幂运算的幂。d的值是使得对于任何不整除n的m，我们有（m^e）^{d}和$m \bmod n$同余的性质，类似地，（m^d）^{e}也和同样的值同余。

e和n形成公钥，而d和n形成私钥。一个成员，给定e和n，他不能轻易地计算出d，或者轻易地求出$c = m^e \bmod n$这个计算式的逆。

对于怎样选择素数的细节，读者可以参考各种介绍，例如*BigNum Math*，(Tom St Denis，Greg Rose，*BigNum Math——Implementing Cryptographic Multiple Precision Arithmetic*，Syngress，2006)，它深入地讨论了这方面的知识。该书也讨论了快速RSA运算所要求的其他方面的知识，例如快速模幂运算。读者也可以考虑*The Art of Computer Programming*（Donald Knuth，*The Art of Computer Programming*，*Volume* 2，第三版，Addison Wesley）作为这个主题的另一个参考。后者是从非常学术的观点来看待这些运算的，而且对于向学生教受有效的多精度算术很有用。

2. RSA变换

RSA变换是通过如下步骤执行的，即把一个消息（解释为一个八位位组的数组）转化成一个整数，用其中一个指数以它进行幂运算，最后把整数转化回一个八位位组的数组。私有的变换是用指数d来对消息进行解密和签名的。公开的变换是用指数e来对消息进行加密和验证的。

9.3.2 PKCS #1

PKCS #1是一个RSA标准，它指定了怎样使用RSA变换来作为一个陷门原型正确地对消息进行加密和签名（PKCS #1可以从www.rsasecurity.com/rsalabs/node.asp?id=2125获得）。用作PKCS #1一部分的填充解决了原始的RSA应用程序具有的各种缺陷。

PKCS #1标准分成4个部分。首先，指定的数据转换算法把消息转化为整数，并且也可以转化回来。接着是密码原型（cryptographic primitives），它是基于RSA变换的，并提供了公钥标准的陷门方面。最后，它用一节来说明一种合理的加密机制，用另一节来说明一种合理的签名

机制。

PKCS #1很明显地使用了ASN.1原型并且需要一个密码散列函数（即使对于加密）。它可以很容易地实现，而不需要一个完整的密码学算法库，这使得它适用于只有有限代码空间的平台。

1. PKCS #1数据转换

PKCS #1指定了两个函数，OS2IP和I2OSP，它们分别执行八位位组串和整数之间的转换。它们用来描述一个八位位组（字节）串是怎样转换成一个用于RSA变换处理的整数的。

OS2IP函数把一个八位位组串映射为一个整数，这是通过用big-endian的方式装载八位位组来完成的。即第一个字节是最有效的。I2OSP函数执行相反的操作而不需要填充。也就是说，如果输入的八位位组串有头零八位位组，I2OSP输出不会影响这一点。我们稍后将会看到PKCS标准是怎样解决这一点的。

2. PKCS #1密码原型

PKCS #1指定了4种密码原型，但是从技术上来看，只有两种惟一的原型。RSAEP原型通过把整数增大为e次方，然后再模n来执行RSA的变换。

RSADP原型执行一种相似的操作，不同的是它使用指数d。除了头零和输入除模数n之外，RSADP函数是RSAEP的逆。这个标准指定了RSADP怎样使用中国剩余定理（Chinese Remainder Theorem, CRT）来加速运算。这并不是数值精确的技术要求，但它通常是一种不错的方法。

这个标准也指定了RSASP1和RSAVP1，分别对应于RSADP和RSAEP。这些原型用于签名算法中，但还不至于特别到我们必须了解它。

3. PKCS #1加密机制

RSA推荐的加密机制叫做RSAES-OAEP，它是OAEP填充和RSAEP原型的简单组合。OAEP填充是提供了抵抗各种活动的敌对攻击的安全机制。解密首先是应用RSADP，然后是OAEP填充的逆操作（如图9-5所示）。

散列函数在标准中没有指定，用户可以自由选择一个。MGF函数的定义如图9-6所示。

RSA的解密首先是应用RSADP，然后是OAEP填充机制的逆运算。如果它们丢失任何常数字节、*PS*串或者*lHash*串，解密都必须拒绝执行。

提示 RSA OAEP填充对能够加密的明文大小做了限制。通常，这算不上是一个问题，因为使用了混合模式，但是，还是值得了解它的。这个限制是由RSA模数的长度和所选的散列函数产生的消息摘要的大小来定义的。

对于RSA-1024——即1024位模数的RSA——和SHA-1，OAEP的负载限制是86个八位位组。对于同样的模数和SHA-256，其限制是62个字节。其限制一般为$k-2*hLen-2$。

对于混合模式，这暴露出了一点问题，因为相应于一个256位的AES密钥，最大的负载将是32个字节。

4. PKCS #1签名机制

类似于加密机制，签名机制在使用RSA原型之前利用了一种填充算法。在签名机制中，这个填充算法称为PSS，机制称为EMSA-PSS。

输入:

(n, e):	公钥，*k*表示*n*以八位位组来表示的长度
M:	加密的消息，长度为*mLen*个八位位组
L:	可选的标签（盐渍），如果没有提供就为空串
hLen:	选择的散列算法产生的消息摘要的长度

输出:

C:	密文

1. if mLen > *k*-2*hLen-2 那么输出“message too long”并返回
2. lHash = hash(*L*)
3. 令*PS*是一个*k-mLen-2*hLen-2*个零八位位组的字符串
4. 连接*lHash*, *PS*, 单独的八位位组0x01和M到DB中
5. 生成一个长度为*hLen*个八位位组的随机串种子
6. *dbMask* = MGF(seed, *k* − *hLen* − 1)
7. *maskedDB* = DB XOR *dbMask*
8. *seedMask* = MGF(*maskedDB*, *hLen*)
9. *maskedSeed* = seed XOR *seedMask*
10. 把单独的八位位组0x00，*maskedSeed*和*maskedDB*连接到EM中
11. *m* = OS2IP(*EM*, *k*)
12. *c* = RSAEP((*e*, *n*), *m*)
13. *C* = I2OSP(*c*, *k*)
14. return *C*

图9-5 带有OAEP的RSA加密机制

输入:

mgfSeed:	掩码（mask）生成种子数据
maskLen:	需要的掩码数据的长度

输出:

mask:	输出掩码

1. 令*T*为空串
2. For *counter* from 0 to ceil(*maskLen* / *hLen*) − 1 do
    1. *C* = I2OSP(*counter*, 4)
    2. *T* = *T* || hash(*mgfSeed* || *C*)
3. 返回*T*开头的*maskLen*个八位位组

图9-6 MGF

EMSA-PSS签名算法的定义如图9-7所示。

输入：

(n, d):	RSA私钥
M:	待签名的算法
emBits:	模的位数
emLen:	等于ceil（embits/8）
sLen:	盐渍的长度

输出：

S:	签名

1. *mHash* = hash(*M*)
2. If *emLen* < *hLen* + *sLen* + 2，输出“encode error”并返回
3. 生成一个随机串*salt*，长度为*sLen*个八位位组
4. *M'* = 0x00 00 00 00 00 00 00 00 || *mHash* || *salt*
5. *H* = Hash(*M'*)
6. 生成一个八位位组串*PS*，由*emLen-sLen-hLen*-2个零八位位组组成
7. DB = *PS*||0x01||*salt*
8. *dbMask* = MGF(*H*, *emLen* − *hLen* − 1)
9. *maskedDB* = *DB* XOR *dbMask*
10. 把*masked D B*最左端的八位位组最左端的8**emLen-emBits*个位设置为0
11. EM = *maskedDB* || *H* || 0xBC
12. s = OS2IP(EM, *emLen*)
13. s' = RSASP1((*d*, *n*), *s*)
14. S = I2OSP(*s'*, *emLen*)
15. return *15*

图9-7 使用PSS的RSA签名机制

验证签名时，首先对签名应用RSAVP1，它返回*EM*的值。然后我们看0xBC八位位组并检查最左端的八位位组的高8**emLen-emBits*位是否为0。如果这两个测试有一个失败，那么这个签名就是无效的。这时，我们从*EM*中提取*maskedDB*和*H*，重新计算*dbMash*并把*maskedDB*解码为*DB*。这时*DB*应该包括原始的*PS*零八位位组（所有的*emLen*−*sLen*−*hLen*−2个项），0x01八位位组和盐渍。从盐渍中，我们可以重新计算*H*并把它和从*EM*中提取的*H*进行比较。如果它们相符，这个签名就是有效的，否则无效。

安全注意事项 *确保在执行RSA原型之后解码了的串和模数的规模是相同的，而且不能小于模数的规模是很重要的。PSS和OAEP算法假设解码了的值和模的规模相同，否则它们不能正确地执行。在解码之后最好用一个测试来检查其长度，如果不正确的话就终止执行。*

5. PKCS #1密钥格式

PKCS #1针对RSA密钥定义了两种密钥格式：一种用于公钥，另一种用于私钥。公钥格式

如下。

```
RSAPublicKey ::= SEQUENCE {
   modulus            INTEGER, – n
   publicExponent     INTEGER, – e
}
```

私钥格式如下。

```
RSAPrivateKey ::= SEQUENCE {
   version            Version,
   modulus            INTEGER, – n
   publicExponent     INTEGER, – e
   privateExponent    INTEGER, – d
   prime1             INTEGER, – p
   prime2             INTEGER, – q
   exponent1          INTEGER, – d mod (p - 1)
   exponent2          INTEGER, – d mod (q - 1)
   coefficient        INTEGER, – (1/q) mod p
   otherPrimeInfos    OtherPrimeInfos OPTIONAL
}
Version ::= INTEGER { two-prime(0), multi(1) }
OtherPrimeInfos ::= SEQUENCE SIZE(1..MAX) OF OtherPrimeInfo
OtherPrimeInfo ::= SEQUENCE {
    prime             INTEGER, – ri
    exponent          INTEGER, – di, d mod prime
     coefficient      INTEGER, – ti
}
```

私钥存储了用来加速私钥运算的CRT信息。它保存在SEQUENCE（序列）中，是不可选的，但是你是否用它是可选的。这个格式也允许由一个值为1的Version并且提供了OtherPrimeInfos来表示的多素数（multi-prime）RSA。支不支持它是可选的，而且一般来说忽略这一点是个不错的方法，因为它由HP专利所有。

9.3.3 RSA的安全

RSA的安全主要取决于把模数分解为构成它的两个素数的能力。如果一个攻击者可以分解这个模数，他就可以计算私有指数并且利用这个系统。一直以来，人们对代数数论领域进行了广泛的研究以发现分解的算法。第一个有用的算法是由Carl Pomerance发明的二次筛法。

二次筛法对于整数n的时间复杂度为O（exp（sqrt（log n log log n）））。例如，分解一个1 024位的数，预期的平均运行时间将是以2^{98}为阶的运算。它通过收集$X^2 = Y$（mod n）的形式关系，并且使用一个叫做因子范围（factor bound）的小的素数集来对Y进行分解。对这些关系进行收集和分析，寻找组合关系使得它们的乘积在两者都为一个平方。例如，如果$X_1^2 * X_2^2 = Y_1Y_2$（mod n）且Y_1Y_2是一个平方，那么这对组合就可以用来进行一次分解。如果我们令$P = X_1^2 * X_2^2$，$Q = Y_1Y_2$，那么如果$x_1 * x_2$和sqrt(Q)模n不同余，我们就可以分解n。如果$P = Q$(mod n)，那么$P - Q = 0$(mod n)，而且因为P和Q都是平方，所以可能用其平方差来分解n。

一种较新的叫做数域筛的分解算法试图构造相同的关系，但是以非常不同的方式。它起初是用来解决一种特殊形式的数（非RSA），后来才被改进为通用数域筛。它的时间复杂度为$O(\exp(64/9 * \log(n)^{1/3} * \log(\log(n))^{2/3}))$。例如，分解一个1 024位的数，其预期的平均时间将为

以2^{86}为阶的运算（如表9-1所示）。

这意味着，如果你想花费2^{112}的工作来攻破你的密钥，那么你必须使用至少2 048位的RSA模数。实际上，一旦你使用2 048位，这就标志着你的算法效率会变得非常低，通常不可能在嵌入式系统中实现。也没有在强度方面能够和AES-256或者SHA-512相符的RSA密钥规模。为了从RSA中获得256位的安全，你可能需要一个13 500位的RSA模数，即使对于最快的桌面处理器这也是一种相当重的负担。让分解能实用的一个最大的障碍是所需的规模。一般来说，数域筛需要复杂度平方根一半的存储空间来工作。例如，可能会需要一个含有2^{43}个元素的表来分解一个1 024位的合数。如果每个元素都是一个单独的位，这也将是一个TB级的存储空间，虽然这看起来很多，但当你意识到为了让这个算法更有效的工作，它必须是随机访问的内存，而不是固定的磁盘存储。

表9-1 RSA密钥强度

RSA模数的规模（位）	对于分解的安全性（位）
1024	86
1536	103
1792	110
2048	116
2560	128
3072	138
4096	156

在实际应用中，目前RSA-1024仍然是安全的，但新的应用程序真的应该使用至少RSA-1536或者更高的——尤其是一个将要使用几年的密钥。业界的许多人设置最小为2 048位的密钥，来避免将来可能的实际进展所带来的安全危机。

9.3.4 RSA参考资料

有许多办法可以让模幂算法变得更快。读者将寻求的第一件事就是使用中国剩余定理（CRT），这使得私钥的运算可以分成两个只有原来一半规模的模幂运算。PKCS #1标准解释了怎样使用RSADP和RSAVP1原型来实现CRT幂运算。

在这个优化之后，下一个优化是选择一个合适的幂运算。最基本的是*square and multiply*（参见*BigNum Math*第192页图7-1）算法，它使用最小数量的内存，但花费将近最大数量的时间。从技术上来看，*square and multiply*容易受到时间攻击（timing attack），因为它只是做乘法运算，直到在幂中有一个相符的位。*Montgomery Powering Ladder*方法可以避免这种危险，但也可能是最慢的幂算法（Marc Joye and Sung-Ming Yen，"The Montgomery Powering Ladder"，Hardware and Embedded Systems, CHES 2002, vol. 2523 of Lecture Notes in Computer Science, pp.291-302, Springer-Verlag,2003）。不像*blinded exponentiation*技术，*Montgomery powering ladder*是完全确定的，而且在运行时并不需要熵来执行运算。这个算法如图9-8所示。

这个算法执行的是相同的运算，至少在高层次上是，而不管幂的位是什么——考虑到敌人可以收集侧信道信息，这可能是各种威胁模型的关键。如果认识到每次迭代的两个运算可以并行的话，那么它也可以提高运算速度。这使得硬件实现可以以牺牲存储空间为代价来换取一定的速度。这个观测对椭圆曲线算法更有好处，正如我们稍后将看到的，因为这些数更小，并且硬件也可以更小。

输入：

g: 底

k: 幂，用位表示为（$k_{t-1}, \ldots, k_0$）$_2$

输出：

y: $y = g^k$

1. $R_0 = 1, R_1 = g$
2. for $j = t - 1$ downto 0 do
   1. if ($k_j = 0$) then
   $R_1 = R_0 * R_1, R_0 = R_0^2$

else

$R_0 = R_0 * R_1, R_1 = R_1^2$

3. Return R_0

图9-8 Montgomery Powering Ladder

一种简单的盲化（blinding）技术是对一个随机的r预计算g^{-r}，并且用$g^{k+r} * g^{-r}$来计算g^k。只要你从不重用r，它是随机的而且和k的长度相同，那么这种技术就可以盲化可能会被k泄露的时间信息。

如果内存充足，窗口化的幂算法也是一种可能的选择办法（参见*BigNum Math*第196页图7-4）。它允许把几个乘法运算组合到一步中，但却是在花费预计算时间和存储空间的情况下。它可以通过仔细地放置一些假的运算而修改成能在固定的时间内运行。

9.4 椭圆曲线密码学

在过去的20多年中，一种不同形式的公钥密码逐渐发展壮大——椭圆曲线密码学，或者叫ECC。ECC围绕一种通常很难理解的代数运算集来执行密码学运算，它比RSA更快，更安全。

ECC使用称为一个椭圆曲线的数学结构来构造一个陷门，它是一种不容易受子幂攻击的方式（目前而言）。这意味着用于ECC的每个位都趋向于终结这个原型的安全性而不像在RSA中那样。由于这个原因，用于ECC的数（或者多项式）比用于RSA中的要小得多。这使得其整数运算更快而且使用更少的内存。

针对本书的目的，我们将讨论称作素数域的ECC曲线，它由NIST定义并且用于NSA Suite B协议。NIST也定义了一个二进制域的ECC曲线集，但它们不太适合于软件实现。对这两种情况，一个极好的资源是书*Guide to Elliptic Curve Cryptography*（Darrel Hankerson，Alfred Menezes，Scott Vanstone，*Guide to Elliptic Curve Cryptography*（中文译为《椭圆曲线密码学导论》），Springer，2004）。这本书详细地描述了二进制和素数域曲线的ECC数学的软件实现。如果读者对实现ECC感兴趣，强烈建议阅读这本书，因为它会给出这里没有囊括的高度优化以及实现上的知识点。

9.4.1 什么是椭圆曲线

一个椭圆曲线典型地是由一个立方等式的平方根定义的一种两空间的图形。例如，$y^2 = x^3-$

x是一个在实数集上的椭圆曲线。椭圆曲线也可以定义在其他域上，例如，模一个素数的整数域，表示为$GF(p)$，以及各种基的扩展域，例如$GF(2^k)$（这称为二进制域ECC）。

给定一个定义在模p的有限域上的椭圆曲线，例如$E_p: y^2 = x^3-3x + b$（典型的素数域定义），我们可以计算在这个曲线上的点。一个点就是一个满足曲线等式的二元组（x,y）。因为在个域中有有限个数，所以在这个曲线上必定有一个有限的惟一的点的数目。这个数叫做这个曲线的阶（order）。由于各种密码学上的原因，我们要求其阶是一个大素数，或者它的因子中有一个大的素数。

NIST指定了5个椭圆曲线，其域的大小分别为192、224、256、384和512位。这5个曲线本身的阶都是大素数，它们都满足前面列出的E_p的定义，不同的是b的值各不相同。它们具有相同的曲线方程是很重要的，因为它使得我们可以使用相同的基本数学来处理这些曲线。仅有的区别是，模数和数的规模发生了改变（已证明你并不需要b来执行ECC运算）。

从我们的椭圆曲线中，可以构造出一种代数，其运算有点加、倍点和点乘。这些操作使得我们可以创建一个对DSA签名和基于Diffie-Hellman的加密有用的陷门函数。

9.4.2 椭圆曲线代数

椭圆曲线拥有一种代数，它允许用一定的方式对曲线上的点进行操作。点加是取曲线上的两个点并构造另一个。如果我们从和中减去一个原始的点，那么我们就可以计算出另一个原始点。倍点取一个单独的点并计算这个点自身相加等于什么。最后，点乘组合了前两种运算并使得我们对一个点乘以一个标量。我们稍后将会看到，是最后一个运算创建了陷门。

1. 点加

点加定义为计算两点之间的斜率并找出从这两点中的任一点开始与曲线相交的地方。通过对第三个点的y部分取反来将它取反以计算这个加法。给定曲线上两个不同的点$P = (x_1, y_1)$和$Q = (x_2, y_2)$，我们可以用如下的公式来计算其点加。

$$P + Q = (x_3, y_3)$$
$$x_3 = ((y_2 - y_1) / (x_2 - x_1))^2 - x_1 - x_2$$
$$y_3 = ((y_2 - y_1) / (x_2 - x_1)) * (x_1 - x_3) - y_1$$

所有这些运算都是在有限域上执行的，即以某个素数为模。当P和Q的x坐标不相同时，这些等式才有定义。

2. 倍点

倍点定义为计算一个点对于曲线的正切并找出它与曲线的交点。其交点应该是惟一的，倍点就是这个点的两倍。倍点可以认为是一个点加上它自身。通过使用正切而不是斜率，我们可以得到明确定义的行为。给定$P = (x_1, y_1)$，其倍点可以如下计算。

$$2P = (x_3, y_3)$$
$$x_3 = ((3x_1^2 + a) / (2y_1))^2 - 2x_1$$
$$y_3 = ((3x_1 + a) / (2y_1)) * (x_1 - x_3) - y_1$$

这种运算也是在一个有限域上执行的。a的值取自曲线的定义，对于NIST的曲线它等于-3。

3. 点乘

点乘定义为一个点对它自身相加的数次，一般表示为kP，其中k是我们想让P加上自身的次数的标量数值。例如，如果我们写$3P$，其字面意思等价于$P+P+P$，或者更特别地讲是一个倍点和一个加法。

提示 一开始很容易对椭圆曲线数学感到困惑不解。其符号对于大多数的开发者而言是非常新的，而且所要求的运算更为奇怪。一直以来，椭圆曲线资料的大部分作者都试图使他们的符号表示一致。

当某人读到"kG"，小写字母几乎总是标量，大写字母是曲线上的点。字母k有时候也用来表示随机的标量，例如，Diffie-Hellman协议就需要这样定义。字母G有时候也表示曲线上的标准的基点。

在使用点乘的过程中，曲线的阶发挥了很重要的作用。阶指定了最大的惟一点的数目，它给出了曲线上一个理想的固定点G。也就是说，如果我们循环遍历所有可能的k值，我们将碰到所有的这些点。对于NIST指定的曲线，其阶都是素数，这有一个很好的副作用，即曲线上所有的点都有最大的阶。

9.4.3 椭圆曲线加密系统

利用椭圆曲线来构造一个公钥加密系统，我们需要一种创建一个公钥和私钥的方法。对此，我们使用点乘函数和一个标准指定的位于曲线上的基点，并对所有的用户共享。NIST对他们支持的5个椭圆曲线每个都提供了一个基点（在ECC的素数域上）。

1. 椭圆曲线参数

NIST指定了5个素数域曲线用于加密和签名算法。这个标准的一个PDF拷贝可以从NIST获得而且很便于随时阅读（NIST推荐的曲线：http://csrc.nist.gov/CryptoToolkit/dss/ecdsa/NISTReCur.pdf）。

NIST指定了5个规模分别为192、224、256、384和512位的曲线。当我们说规模，实际上是说曲线的阶。例如，对于192位的曲线（也叫做P-192），其阶是一个192位的数。对于素数域上的任一个曲线，可以用4个参数来描述这个曲线。第一个是模数p，它定义了曲线所在的域。下一个是参数b，它定义了在有限域中的曲线。最后是曲线的阶n和在曲线上具有特定阶的基点G。元组(p, n, b, G)是一个开发者用来实现椭圆曲线加密系统所全部需要的。

为了完整起见，下面是P－192曲线设置，因此你可以看到它们是什么样子的。我们不列出全部的5个，因为打印一页看起来全是随机的数没有什么用。可以考虑阅读NIST提供的PDF或者查看LibTomCrypt中的src/pk/ecc/ecc.c。这个文件用可以很容易嵌入到其他应用程序的格式包含了所有的5个曲线（LibTomCrypt是公开的）。如下的代码是从这个文件截取的一个片段。

```
#ifdef ECC192
{
        24,
        "ECC-192",
```

```
        "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFFFFFFFFFFFF",
        "64210519E59C80E70FA7E9AB72243049FEB8DEECC146B9B1",
        "FFFFFFFFFFFFFFFFFFFFFFFF99DEF836146BC9B1B4D22831",
        "188DA80EB03090F67CBF20EB43A18800F4FF0AFD82FF1012",
        "7192B95FFC8DA78631011ED6B24CDD573F977A11E794811",
},
#endif
```

这个结构的第一行用字节的形式来描述域的规模。第二行是对曲线字面上的命名（用于诊断的支持）。然后是p, b, r，最后是基点（x, y）。所有的数以十六进制格式存储。

2. 密钥生成

图9-9中的算法描述了密钥生成处理过程。

输入：

G: 标准基点

n: 曲线的阶（NIST提供）

输出：

Y: 公钥（ECC的点）

x: 私钥（整数）

1. 取一个随机整数x，$1<x<n-1$
2. 计算$Y=xG$
3. 返回（x, Y）

图9-9 ECC密钥生成

现在我们可以把Y发送给任何想要的人，接收者可以用它对消息进行加密，或者对我们签名的消息进行验证。其格式——即我们怎样存储公钥的字节格式——被指定为ANSI X9.63的一部分（4.3.6节），但是，还有更好的格式。出于方便读者的目的，我们将讲解ANSI的方法。

3. ANSI X9.63密钥存储

一个椭圆曲线上的点可以用ANSI标准中的3种形式之一来表示。

- 压缩的（Compressed）；
- 非压缩的（Uncompressed）；
- 混合的（Hybrid）；

当存储$P = (x, y)$时，首先把x转化成一个八位位组串X。对于素数域曲线，这意味着是把这个数存储为一个big-endian格式的八位位组数组。

- 如果使用压缩的形式，那么
  - 计算t = compress(x, y)
  - 当t为0时，其输出为0x02 ‖ X；否则，其输出为0x03 ‖ X
- 如果使用非压缩的形式
  - 把y转化为八位位组串Y
  - 输出为0x04 ‖ X ‖ Y

- 如果使用混合形式
  - 把y转化成一个八位位组串Y
  - 计算t = compress（x, y）
  - 当t为0时，输出为0x04 ‖ X ‖ Y；否则，其输出为0x05 ‖ X ‖ Y

压缩函数计算y的平方根。按照ANSI X9.63（4.2.1节），我们可以通过取y的最右边位来压缩一个点（x, y）并返回它作为输出。对这个点进行解压缩是相当简单的，给定x和压缩位t，解压缩为：

计算a，它是$x^3-3x+b \pmod p$的平方根

如果a最右边的位等于t，那么就返回a；否则返回$p-a$

注意 读者应该慎重使用点压缩。Certicom拥有美国专利号#6、141、420，它描述了把一个完整的点压缩为只剩x坐标和一个附加位的方法。因为这个原因，通常避免使用点压缩。

一般来说，ECC的密钥是如此的小以至于并不需要点压缩。但是，如果你仍然想压缩你的点，有一个比较聪明的技巧，你可以用它来避免Certicom的专利（如图9-10所示）。

输入：

G: 标准基点

p: 模数

n: 曲线的阶

输出：

k: 秘密密钥（整数）

x: 公钥（仅返回公钥的x坐标，整数）

1. 取一个随机整数k，$1<k<n-1$
2. 计算$Y=kG=(x, y)$
3. 计算$z=sqrt(x^3-3x+b) \pmod p$
4. 如果x不等于y，转到第1步
5. 返回（k,x）

图9-10 带有点压缩的椭圆曲线密钥生成

这种密钥生成方法几乎比普通密钥生成算法慢两倍，但它能生成你可以压缩的公钥并避免了Certicom的专利。由于压缩的位总是0，所以不需要对它进行传输。用户只需要发送其公钥的x坐标。同时，这种密钥生成方法所产生的密钥可以和其他像EC-DH及EC-DSA的算法兼容。

4. 椭圆曲线加密

用椭圆曲线来加密是使用了ElGamal算法的一种变体，它在ANSI X9.63的5.8节中定义。该算法如图9-11所示。

ANSI在加密中允许额外的共享数据，而且这是可选的。如果没有可选的共享数据的话，其密钥衍生函数等价于PKCS #1标准中推荐的掩码生成函数（Mask Generation Function，MGF）。

所需的MAC（第6步）必须是一个ANSI认可的MAC，安全级别至少为80位。他们推荐X9.71，它是HMAC的ANSI标准。HMAC-SHA1能提供X9.63所要求的最低安全性。

输入：
  Y: 接收者的公钥
  EncData: 明文，长度为*encdatalen*
  mackeylen: 所需的MAC密钥的长度
输出：
  c: 密文

1. 生成一个随机密钥$Q=(kq, (x_q, y_q))$
2. 计算共享秘密z，k_qY的x坐标
3. 利用密钥衍生算法（ANSI X9.63第5.6.3节，等价于PKCS #1 MGF）把z变换为KeyData串，其长度为encdatalen+mackeylen位
4. 把KeyData分成两个串，EncKey和MacKey，长度分别为encdatalen和mackeylen位
5. MaskedEncData = EncData XOR EncKey
6. MacTag=MAC（MacKey,MaskedEncData），使用一个ANSI认可的MAC算法（例如X9.71,即HMAC）
7. 使用密钥存储算法存储公钥(x_q,y_q)，称之为QE
8. 返回

图9-11 椭圆曲线加密

解密过程为计算$k(x_q, y_q)$的x坐标为z，其中k是接收者的私钥。利用z，接收者能够生成加密和MAC密钥，校验MAC并解密密文。

5. 椭圆曲线签名

椭圆曲线签名是一种称为EC-DSA的算法，它衍生自NIST的标准数字签名算法（DSA，FIPS-186-2）和ElGamal。签名算法在ANSI X9.62的第7.3节（如图9-12所示）。

这个签名实际上是两个和阶有同样规模的整数。例如，对于*P*-192，签名大概为384位的大小。ANSI X9.62指定（E.8节）当传输签名时，以如下的ASN.1 SEQUENCE存储。

```
ECDSA-Sig-Value ::= SEQUENCE {
   r    INTEGER,
   s    INTEGER
}
```

考虑到它是可惟一解码的，所以它是一个来自X9.63的受欢迎的改变。验证算法如图9-13所示（ANSI X9.62 7.4节）。

为了进一步的澄清，第7步执行的是两个点乘之后的相加。

提示 有两种办法来加速验证。第一种是使用被称为“Shamir's Trick”的方法来计算*R*。你可以在“Guide to Elliptic Curve Cryptography”第109页的算法3.48中找到它的描述。这可以让你即遵循了标准又可以用更少的时间来计算*R*。

如果你将要在一个资源紧张的平台上执行这个认证并且能忍受对标准的一些偏

离，你可以把签名创建为（r, $1/s$）。这可以省略让签名生成更慢所需要的模逆运算。但是，这不是ANSI所允许的。

输入：

k: 私钥（整数）

n: 曲线的阶

M: 待签名的消息

输出：

(r, s): 签名

1. 生成一个随机密钥$Q=$（k_q,（x_q,y_q））
2. 把x_q转化为一个整数j。对于素数曲线这一步可以省略，因为x_q已经是一个整数
3. $r = j$ mod n，如果$r = 0$，转到第1步
4. H = hash（M）
5. 截短H到最左端的ceil（log2n）位。例如，对于P-192，截短H到192位
6. 通过以big-endian的格式来加载H，把它转化为一个整数e
7. 计算$s=k_q-1$（$e + kr$） mod n; 如果s=0，转到第1步
8. 返回（r, s）

图9-12　椭圆曲线签名

输入：

Y: 签名者的公钥

G: 曲线的基点

M: 已签名的消息

n: 曲线的阶

(r, s): 签名

输出：

Validity of signature 签名的有效性

1. 如果r和s都不在区间[1,n−1]中，那么就返回invalid（无效）
2. H=hash(M)
3. 截短H到最左端的ceil（$\log_2 n$）位。例如，对于P-192，截短H到192位
4. 通过以big-endian的格式来加载H，把它转化为一个整数e
5. $u1=e/s$ mod n
6. $u2=e/s$ mod n
7. $R = u_1G + u_2Y = (x_r, y_r)$
8. 把x_r转化为一个整数j。对于素数曲线这一步可以省略，因为x_r已经是一个整数
9. $r' = j$ mod n
10. 如果r等于r'，返回valid（有效）；否则，返回invalid（无效）

图9-13　椭圆曲线签名认证

PDG

9.4.4 椭圆曲线的性能

目前为止，我们只看了最基本的椭圆曲线运算，例如使用仿射坐标的点加和倍点。如果我们用目前为止所描述的函数来实现一个加密系统，那将是非常慢的。这是因为模逆——即计算以一个素数为模的$1/x$——是一种非常慢的运算，常常比得上许多个域乘法运算。

为了解决这个问题，人们描述了各种投影系统，把两空间的运算映射到三空间或者更多的空间上去。其目标是用域上的乘法来代替模逆。理想情况下，如果将域乘法减到最小，那么你就可以实现平衡而且使得点加和倍点运算更快。照此下去，它也会使得点乘运算变得更快，最后是独立的操作例如加密和签名。

另一个重要的优化是我们怎样执行点乘自身。如果我们只是简单地把点自身相加必需的次数，这会花很多时间来执行一次点乘。也有各种针对它的窗口和定点算法。

1. Jacobian投影点

在Jacobian投影坐标系统下，我们用3个变量来表示一个标准点。转化为投影格式的映射非常简单。前向映射是从$z = 1$开始，将(x, y)映射为(x, y, z)。如果从投影坐标映射回来，我们只需要计算$(x/z^2, y/z^3, 1)$，其中的除法是在有限域中执行的。特别地，要先计算$1/z$，并从它衍生出$1/z^2$和$1/z^3$，这只需要一次模逆运算。

例如，倍点就变成了如图9-14所示的算法（取自“Guide to Elliptic Curve Cryptography”第91页算法3.21）。点加在同一页中，列为算法3.22。

```
输入：
    P:              Jacobian坐标下的 (x1, y1, z1)
输出：
    (x3, y3, z3):   2P的值

t1 = z1^2
t2 = z1 - t1
t1 = x1 + t1
t2 = t2 * t1
t2 = 3 * t2
y3 = 2 * y1
z3 = y3 * z1
y3 = y3^2
t3 = y3 * x1
y3 = y3^2
y3 = y3/2
x3 = t2^2
t1 = 2 * t3
x3 = x3 - t1
t1 = t3 - x3
t1 = t1 * t2
y3 = t1 - y3
Return (x3, y3, z3)
```

图9-14 Jacobian投影倍点

其中除以2的值需要是偶数，否则你必须首先加上模数（强制它为偶数），然后再执行这个除法（通常是通过执行右移来实现）。正如我们所看到的，这个倍点比原始的倍点算法的步骤稍微多点。实际上，它也有更多的域乘法运算。但是，它没有模逆，而且大部分的运算都是简单的加法或减法。

点加算法是假设第二个输入是在仿射坐标下的（z=1），所以记住什么时候执行点乘是很重要的。如果你不想映射到仿射坐标，那么你可以使用由T. Hasegawa，J. Nakajima和M.Matsui所写的“A practical Implementation of Elliptic Curve Cryptosystems over Gf（p） on a 16-bit Microprocessor”中4.2节的算法。它描述了Jacobian-Jacobian以及Jacobian-affine两种技术。混

合的加法稍微有点快而且消耗更少的内存（因为第二个操作数只需要两个坐标的存储空间），但是在初始化时消耗比较多。

2. 点乘算法

有多种有用的方法可以完成点乘运算，但最基本的是double and add方法。它实质上是幂运算中的square and multiply转换到点乘上的（Guide一书中的算法3.27，第97页）。

这种方法虽然有效，但不经常使用，因为它太慢了。大多数的开发者选择滑动窗口方法（Guide一书中的算法3.38，第101页）。它需要更少的点加而且消耗很少的内存。这个算法类似于*BigNum Math*第198页中的图7-7的幂运算技巧。

另外一种用于当点是已知的（或固定的）情况的方法是定点技术。这个算法的一种变形在Guide一书的第104页中算法3.41进行了描述。它也是最近才加到LibTomCrypt中的，但也有一些稍微的变形。这种技术可以产生很快的点乘，但是以牺牲内存为代价。它对加密、签名生成和验证都很有用。它不能用于解密，因为其点通常是随机的。因此在一个快速的定点乘法函数上花费资源之间，有一个快速的随机点乘法函数（当解密需要时）更为重要。

Guide这本书还描述了各种基于Nonadjacent范式（NAF）编码的方法，它对于内存有限的平台似乎是很理想的。在实际中，它们常常没有普通的窗口方法快。但是，如果内存是一个问题的话，它很值得去研究。

9.5 总结

在各种标准中，有两种可以胜任的算法：RSA，它相对比较旧而且更传统地用于业界；ECC，稍微有点新，更加有效，而且产生了一种新的局面。

如果没有给定逻辑上的限制，例如遵循的标准，那么在RSA和ECC之间进行选择通常是很容易的。

9.5.1 ECC与RSA

1. 速度

使用Jacobian坐标和一个合理的点乘函数的ECC和RSA比较，可以产生更快的私钥运算。RSA使用更大的整数和必须执行一个慢的模幂运算的事实使得任何私钥运算都是非常慢的。通过使用更小的整数，ECC可以更快地执行它的原型运算，例如域乘法。

RSA胜过ECC的地方是在公钥运算中（例如加密和认证）。因为其公开指数e通常是很小的，其模幂运算可以很有效地计算出来。例如，当$e = 65\ 537$时，只需要16个域上的平方和1个域上的乘法运算来完成这个幂运算。虽然它的域更大，但它最终还是取得了胜利。

在表9-2中，在AMD Opteron上的RSA-1024的加密需要131 176个时钟周期，而使用ECC-192（在位强度上最接近的ECC）的加密需要两个点乘运算，而且将消耗至少780 000个时钟周期。另一方面，RSA-1024的签名生成需要至少120万个时钟周期，而ECC-192只需要390 000个时钟周期。

表9-2 公钥算法的性能（每个运算的时钟周期）

PSA Tests (Encrypt/Decrypt)	AMD Opteron	Intel Pentium 4	Intel Pentium M	Intel Core 2 Duo
RSA-1024	131,176/1,243,530	694,262/7,875.960	467,330/4,851,959	146,020/1,549,781
RSA-1536	225,191/3,562,010	1,090,842/13,470,711	933,843/13,816,129	254,741/4,314,161
RSA-2048	348 006/6,144,918	2,143,433/47,124,161	1,495,344/29,705,622	396,468/8,294,762
ECC Tests 定点乘法运算				
ECC-192	390,615	1,470,551	989,938	367,879
ECC-224	468,685	1,952,344	1,280,312	454,996
ECC0256	583,723	2,464,947	1,547,887	586,797
ECC-384	1,123,152	5,916,616	3,572,755	1,240,034
ECC-521	1,986,757	12,877,997	7,743,301	2,304,239

2. 大小（规模）

ECC使用更小的数（其原因稍后将出现），这使得在传输数据时它使用更少的内存和存储空间。

例如，当目标是80位的安全性，ECC只要使用一个160位的曲线就可以了，而RSA却要使用一个1 024位的模数。这意味着我们的ECC公钥只是320位的大小。这也意味着我们的EC-DSA签名也只是320位，相对于RSA-PSS所需要的1 024位。

在实现阶段它也是很有好处的。一个单独的RSA大小的整数所需要的空间可以存储将近8个或者更多的ECC参数。我们可以从Jacobian倍点算法中看出我们需要11个整数（1个用于模数，1个用于溢出，3个用于输入，3个用于输出，3个临时变量）。这意味着对于ECC-192，这个运算只需要264个字节的存储空间，点加需要360个字节。RSA-1 024用一个简单的square and multiply就需要至少4个整数，需要512个字节的存储空间，（在实际应用中，任何让RSA加速的算法会需要更多的整数。这种大小的估计仅仅是——估计）。

3. 安全性

RSA的安全性大部分依赖于因子分解的困难性。虽然这并不完全正确，但是目前都是按照这条线来对RSA的公钥进行攻击的。从另一方面来讲，ECC不容易受因子分解的攻击或其他在子幂范围内的攻击。

假设曲线的阶是素数的情况，仅有的已知针对NIST选择的椭圆曲线攻击是一个幂时间循环查找攻击，称为Pollard-Rho。如果曲线的阶是n，那么将需要将近$O(n^{1/2})$的运算来攻破ECC的公钥并找到其秘密密钥。例如，对于P-192，攻击者将不得不平均花费大概2^{96}次对密钥的攻击。这意味着当对公钥进行离线攻击时，攻破P-192实际上比RSA-1024困难1 024倍。实际上，RSA-1024相应于ECC P-192，RSA-2048相应于ECC P-224，RSA-3072相应于ECC P-256。

这才是ECC真正特别的地方。对于112位的安全级别（RSA-2048），ECC P-192运算并不困难多少。从另一方面来讲，RSA-2048使用起来更加困难。其域操作慢4倍而且比它们多2倍。在AMD Opteron平台上，当从RSA-1024转为RSA-2048时，私钥运算大约慢600%，而当从ECC P-192转为P-224时，点乘运算只变慢20%。

9.5.2 标准

IEEE和ANSI都指定了椭圆曲线密码学的标准。IEEE P1363a是当前的IEEE标准，而X9.62和X9.63是当前的ANSI标准。NIST指定使用ANSI X9.62 EC-DSA作为数字签名标准的一部分（DSS，FIPS 180-2）。

ANSI指定了RSA标准（X9.61），PKCS #1是其他标准默认参考的标准。特别地，许多旧的系统使用PKCS #1 v1.5，应该避免这种情况。这并不是因为PKCS #1 v1.5完全不安全，只是从安全角度来看它没有v2.1更理想。新的应用程序应该避免X9.31（它很相似于PKCS #1的v1.5）和PKCS #1的v1.5，而使用v2.1。

9.5.3 参考资料

1. 参考书目

当实现RSA时，PKCS #1标准（v2.1）是目前为止最重要的资源。它描述了OAEP和PSS填充技术、CRT幂算法，以及针对交互性的ASN.1定义。对于FIPS 180-2 DSS，必须使用ANSI X9.61标准。

当实现ECC时，ANSI X9.62标准指定了EC-DSA并用于FIPS 180-2 DSS中。ANSI X9.63标准指定了ECC加密、密钥存储和许多认证机制（有几个是有专利的）。目前，NIST正在工作于SP800-56A，它指定了使用离散对数系统（类似于ElGamal）的ANSI X9.42，以及使用ECC的X9.63。另外一个SP 800-56B指定了ANSI X9.44（RSA 加密）。很有可能SP800-56A在将来会变得更加流行，因为它使用ECC来代替RSA。

关于大整数运算的一本好的参考书是*BigNum Math*（Tom St Denis, Greg Rose, *BigNum Math: Implementing Cryptographyic multiple Precision Arithmetic*, Syngress, 2006, ISBN 1-59749-112-8），它讨论了一种可移植且高效的多精度大整数运算的创建。这本书使用伪码和真正的C源代码来给读者演示这些数学。它阅读起来绝不困难而且很适合这本书的目标读者。

对于实现ECC数学，强烈建议读者获得一份*Guide to Elliptic Curve Cryptography*（D.Hankerson，A. Menezes，S.Vanstone，*Guide to Elliptic Curve Cryptography*，Springer，2004，ISBN 0-387-95273-X）的拷贝。这本书即讨论了二进制域也讨论了素数域上的ECC运算，并详细地解释了点运算和乘法运算。它在域运算上花了很少的时间（例如整数乘法），但在二进制域运算上花得时间很多（对大多数人来讲，它的运算一开始是很难理解的）。它是在点乘技术上非常珍贵的资源。

2. 参考源代码

LibTomCrypt包提供PKCS #1允许的RSA以及ANSI X9.62允许的EC-DSA。它使用一种修改了的密钥衍生函数和与X9.63不兼容的密钥存储方案。LibTomCrypt对RSA使用了CRT幂运算。对ECC，它使用了Jacobian仿射坐标。它即提供了一个滑动窗口随机点乘函数，也提供了一个定点乘法函数。由于代码的注释很好而且是公开的，所以它也是实现的一个有价值的资源。

Crypto++包也提供了PKCS #1，并支持ANSI X9.62和X9.63。它的基本代码要比LibTomCrypt大而且不是公开的。它也包含多种专利算法，例如EC-MQV。读者应该慎重地使用这个代码，因为它并不是完全免费的而且用起来也不是很方便。

9.6 常见问题

下面的常见问题，由本书的作者所回答，它们即可以用来测试你对本章所出现的概念的理解，也可以帮助你在现实生活中实现这些概念。如果希望作者解答你的问题，请浏览www.syngress.com/solutions然后点击“Ask the Author”表单。

问：什么是公钥算法?

答：公钥算法就是一个密钥可以分成一个公开的和一个私有部分的算法。这使得私钥的持有者能够执行使用公钥所不能执行的运算。例如，私钥可以用来对消息进行签名，而公钥却只能对它们进行验证。

问：它们在其他哪些方面也很有用?

答：公钥密码解决，或者至少给密钥分配提供了一套解决方案，而这个问题一直困扰着对称算法。例如，人们可以用一个公钥对消息进行加密，而且只能用私钥才能解密。使用这种方法，如果一开始没有让一个共享的秘密安全，我们仍然能够传输加密了的数据。

问：有哪些公钥算法?

答：有许多，如Diffie-Hellman、ElGamal、RSA、ECC和NTRU。大多数的标准都是把注意力放在RSA上，而且最近常常转移到ECC上。ElGamal仍然认为是安全的，但它和RSA（现在是专利免费的）相比，效率很低。

问：有哪些公钥方面的标准?

答：ANSI X9.62指定了ECC签名，而ANSI X9.63指定了ECC加密。PKCS #1指定了RSA加密和签名。ANSI X9.31指定了RSA签名，而X9.42指定了RSA加密。FIPS 180-2针对ECC指定了X9.62，对RSA指定了X9.31，以及它自己的DSA（在整数上的）。

问：RSA有哪些优点和不足?

答：从概念上来说，RSA非常简单。它相当容易开发，而且PKCS #1消化起来也不是很困难。RSA公钥运算（加密和认证）非常快。从另一方面来讲，RSA的私钥运算则非常慢。RSA也需要大的密钥用来实现需要的位安全性。例如，对于112位的安全性，人们必须使用一个2048位的RSA密钥。这可以让RSA效率很快就会变得很低。

问：ECC有哪些优点和不足?

答：ECC的安全性比RSA要好得多。虽然使用更小的参数，但ECC比RSA提供了更高的安全。这意味着ECC数学可以高效并快速的存储。ECC公钥运算比相应的私钥运算要慢，但在困难的位安全级别上仍然比RSA快。ECC的实现更为复杂而且更容易出现错误。在概念上它很难变得更很快而且比RSA需要更多的时间来完成。

问：有没有我能够参考的算法库?

答：LibTomCrypt和Crypto++都实现了公钥算法。后者实现了许多的算法，而不仅仅是

RSA和ECC。LibTomCrypt并不完全遵循X9.63标准，因为它做了一些修改以解决X9.63的一些问题。但是，它确实针对RSA实现了X9.62和PKCS #1。

问：我为什么应该购买*Guide to Elliptic Curve Cryptography*?

答：Guide这本书是由一些最好的密码学家所写，他们来自Certicom，而且他们实际上是椭圆曲线数学领域中的专家。这本书也完整地描述了NIST的ECC系列曲线及高效的、专利免费的并且实现容易的算法。在本书重复它的内容看起来有些浪费时间，而且似乎没有起到很好的效果。相信我们，它值得购买。

PDG

［Ｇｅｎｅｒａｌ Ｉｎｆｏｒｍａｔｉｏｎ］
书名＝程序员密码学
作者＝［加］ＴＯＭ ＳＴ ＤＥＮＩＳ ［英］ＳＩＭＯＮ ＪＯＨＮＳＯＮ著 沈晓斌译
页数＝３２８
ＳＳ号＝１１９０８７３５
出版日期＝２００７年０７月第１版
出版社＝机械工业出版社
原书定价＝３９．００
主题词＝密码 理论
参考文献格式＝（加）ＴＯＭ ＳＴ ＤＥＮＩＳ （英）ＳＩＭＯＮ ＪＯＨＮＳＯＮ著；沈晓斌译． 程序员密码学． 北京市：机械工业出版社， ２００７．０７．